Mimer SQL

Documentation Set

Version 11.0

Mimer SQL, Documentation Set, Version 11.0, December 2024
© Copyright Mimer Information Technology AB.

The contents of this manual may be printed in limited quantities for use at a Mimer SQL installation site. No parts of the manual may be reproduced for sale to a third party.
Information in this document is subject to change without notice. All registered names, product names and trademarks of other companies mentioned in this documentation are used for identification purposes only and are acknowledged as the property of the respective company. Companies, names and data used in examples herein are fictitious unless otherwise noted.

Produced and published by Mimer Information Technology AB, Uppsala, Sweden.

Mimer SQL Web Sites:
https://developer.mimer.com
https://www.mimer.com

Contents

Documentation Set

SQL Reference Manual

Programmer's Manual

User's Manual

System Management Handbook

Mimer SQL

SQL Reference Manual

Version 11.0

Mimer SQL, SQL Reference Manual, Version 11.0, December 2024
© Copyright Mimer Information Technology AB.

The contents of this manual may be printed in limited quantities for use at a Mimer SQL installation site. No parts of the manual may be reproduced for sale to a third party.
Information in this document is subject to change without notice. All registered names, product names and trademarks of other companies mentioned in this documentation are used for identification purposes only and are acknowledged as the property of the respective company. Companies, names and data used in examples herein are fictitious unless otherwise noted.

Produced and published by Mimer Information Technology AB, Uppsala, Sweden.

Mimer SQL Web Sites:
https://developer.mimer.com
https://www.mimer.com

Contents

Chapter 1

Introduction

Mimer SQL is an advanced relational database management system (RDBMS) developed by Mimer Information Technology AB.

The main characteristics of Mimer SQL are zero maintenance, small footprint and high performance. These are based on a number of unique technical solutions to handle some of the more complicated functionality that a database management system must provide.

For example, Mimer SQL provides a solution to the problem of allowing simultaneous access to the database without the danger of a deadlock occurring. This greatly simplifies database management and allows truly scalable performance, even during heavy system-load.

Another significant technical innovation is the data storage mechanism, which is constantly optimized for the highest possible performance and ensures that no manual reorganization of the database is ever needed.

Mimer SQL offers a uniquely scalable and portable solution, including multi-core support. The product is available on a wide range of platforms from small embedded and handheld devices running for example Android or Linux, to workgroup and enterprise servers running Linux, Windows, macOS and OpenVMS. This makes Mimer SQL ideally suited for open environments where interoperability, portability and small footprint are important.

The database management language Mimer SQL (Structured Query Language) is compatible in all essential features with the currently accepted SQL standards, see the *Introduction to SQL Standards* on page *9*, for details.

This manual provides a full reference description of the Mimer SQL language. It is a complement to the *Mimer SQL User's Manual* and the *Mimer SQL Programmer's Manual*.

About this Manual

The manual is intended for all users of Mimer SQL in both interactive and embedded contexts. It contains a complete description of the syntax and usage of all statements in Mimer SQL.

Related Mimer SQL Publications

- **Mimer SQL Programmer's Manual**

  contains a description of how Mimer SQL can be embedded within application programs, written in conventional programming languages.

- **Mimer SQL User's Manual**

  is user-oriented guide to the SQL statements, which may provide help for less experienced users in formulating statements correctly (particularly the SELECT statement, which can be quite complex). Does also contain a description of the BSQL utility.

- **Mimer SQL System Management Handbook**

  describes system administration functions, including export/import, backup/restore, databank shadowing and the statistics functionality. The SQL statements which are part of the System Management API are described in the *Mimer SQL Reference Manual*.

- **Mimer SQL Platform-specific Documents**

  containing platform-specific information. A set of one or more documents is provided, where required, for each platform on which Mimer SQL is supplied.

- **Mimer SQL Release Notes**

  contain general and platform-specific information relating to the Mimer SQL release for which they are supplied.

- **Mimer JDBC Guide**

  contains information about the different Mimer JDBC drivers available.

Suggestions for Further Reading

We can recommend to users of Mimer SQL the many works of C. J. Date. His insight into the potential and limitations of SQL, coupled with his pedagogical talents, makes his books invaluable sources of study material in the field of SQL theory and usage. In particular, we can mention the following publication:

A Guide to the SQL Standard (Fourth Edition, 1997). ISBN 0-201-96426-0. This work contains much constructive criticism and discussion of the SQL standard, including SQL-99.

Other useful publications are:

SQL: 1999 - Understanding Relational Language Concepts, by Jim Melton, Alan R. Simon, and Jim Gray. ISBN 1-55860-456-1. Explains SQL-99.

Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features, by Jim Melton. ISBN 1-55860-677-7. In-depth guide to SQL-99's practical application.

Unicode information can be found here https://www.unicode.org.

For JDBC Users

JDBC information can be found here https://www.oracle.com/technetwork/java/index.html.

For information on specific JDBC methods, please see the documentation for the java.sql package. This documentation is normally included in the Java development environment.

JDBC™ API Tutorial and Reference, 2nd edition. ISBN 0-201-43328-1. A useful book published by JavaSoft.

For ODBC Users

Microsoft ODBC 3.0 Programmer's Reference and SDK Guide for Microsoft Windows and Windows NT. ISBN 1-57231-516-4.

This manual contains information about the Microsoft Open Database Connectivity (ODBC) interface, including a complete API reference.

The documentation set in the Mimer SQL Windows distribution includes an ODBC API help.

SQL Standards

Official documentation of the accepted SQL standards may be found in:

ISO/IEC 9075:2016(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL-2016.

Acronyms, Terms and Trademarks

Term	Description
API	Application Programming Interface
BSQL	The Mimer SQL facility for using SQL interactively or by running a command file
CAE	Common Applications Environment
CLI	Call Level Interface
ENV	Européenne Norme Vorausgabe, European Pre-Standard
EOR	European Ordering Rules
IEC	International Electrotechnical Commission
ISO	International Standards Organization
JDBC	The Java database API specified by Sun Microsystems, Inc.
NFC	Normalization Form C
NIST	National Institute of Standards and Technology
ODBC	Open Data Base Connectivity
PSM	Persistent Stored Modules, the term used by ISO/ANSI for stored procedures
SDK	Software Development Kit
SQL	Structured Query Language
UCA	Unicode Collation Algorithm
UCS	Universal Multiple-Octet Coded Character Set

All other trademarks are the property of their respective holders.

Chapter 2

Reading SQL Syntax Diagrams

The syntax of SQL statements is presented in the form of diagrams, showing how the statements may be written. The diagrams are read from left to right.

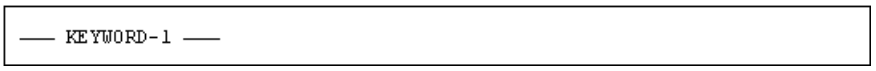
Valid statements are constructed by following the lines in the diagrams and "picking up" elements of the syntax on the way.

It is not practical to give the full, exhaustive syntax of each SQL statement in a single diagram. Instead, many of the syntax diagrams for statements in *Chapter 12, SQL Statements* refer to language elements, which are themselves expanded into syntax diagrams in *Chapter 7, Operators and Values*.

For each syntax diagram, references are given to where in the manual the expansion of the language elements may be found.

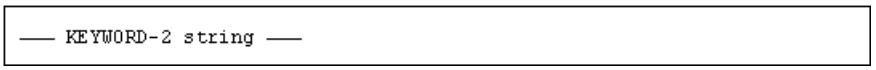
A sample diagram illustrating most of the features of the syntax diagrams is given at the end of this chapter, together with some valid and some invalid formulations.

Key to Syntax Diagrams



A word bounded by diagram lines must be separated from adjoining words by at least one separator.

A separator is represented by a white-space character.



Words separated from each other by at least one space in the syntax diagram must also be separated from each other by at least one separator in the real statement.

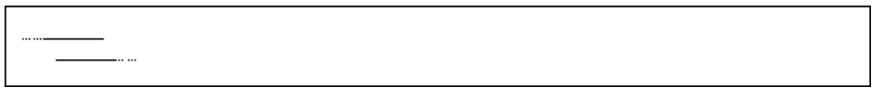
Where the descriptive names for identifiers used in the diagrams consist of more than one word, these are bound together by hyphens.

Branched lines indicate alternative constructions. Only one branch may be followed for any one passage along the line: in this example either `option-1` or `option-2` may be used, but not both.

This representation is used to show that a section of the syntax construction may be repeated. Any construction required between the repetitions is shown on the repeat line.

In this example, the statement must contain at least one instance of `parameter`. If several instances are given, they must be separated from each other by a comma. If a comma or other separator is specified in a list, white spaces need not be used between the elements of the list.

Arrows at the beginning and end of a statement show that the statement is complete.



Dots at the beginning or end of a line in a diagram show that the statement on the line is incomplete.

The continuation may be in the same diagram or relate to a separate diagram, as in the language elements, see *Chapter 7, Operators and Values*. The dots are not part of the statement syntax.

KEYWORDS

Keywords are words that are defined in the SQL language. Keywords are written in `UPPERCASE` in the diagrams. They must always be written in the statement exactly as shown, except that the case of letters is not significant.

Examples of keywords are:

```
ALTER
CREATE
NULL
TABLE
```

Spaces between keywords are significant. Thus the keywords `CREATE TABLE` in this example must be separated by at least one white space character.

Parameters

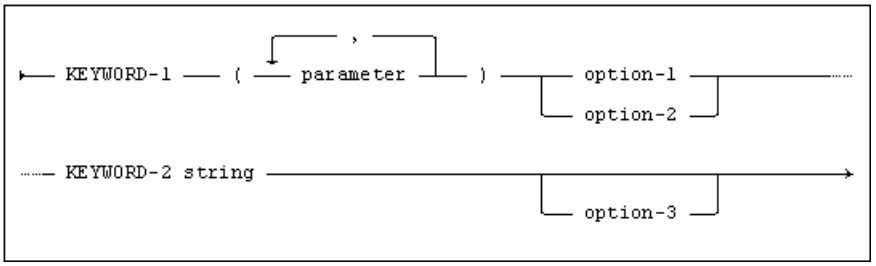
Parameters are indicated by words in lowercase in the diagrams, and replaced by the appropriate identifiers or constructions when statements are written. Examples of parameters are:

```
column-name
expression
data-type
```

The blank spaces in the diagrams are significant. Words bound together by hyphens (e.g. `column-name`, `data-type`) represent single parameters.

Syntax Diagram Example

The following sample illustrates the use of the syntax diagrams.



Some valid formulations are:

```
KEYWORD-1 (parameter) option-1 KEYWORD-2 string

KEYWORD-1 (parameter, parameter) option-1 KEYWORD-2 string option-3

KEYWORD-1 (parameter, parameter, parameter) option-2 KEYWORD-2 string
```

The following formulations are not valid:

```
KEYWORD-1 (parameter) KEYWORD-2 string
```
`option-1` or `option-2` missing

```
KEYWORD-1 parameter  option-1 KEYWORD-2 string
```
parentheses missing

```
KEYWORD-1 (parameter,) parameter option-2 KEYWORD-2 string
```
closing parentheses wrongly placed

```
KEYWORD-1 (parameter, parameter) option-1KEYWORD-2 stringoption-3
```
separating blanks missing

```
KEYWORD-1 (parameter parameter parameter) option-2 KEYWORD-2 string
```
no commas in parameter list

Reading Standard Compliance Tables

For each language element and statement, the standards compliance is noted in a table, e.g. for GRANT ACCESS PRIVILEGE:

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Feature outside core	Feature F731, “INSERT column privileges” support for granting insert on individual columns.
	Mimer SQL extension	The keyword PRIVILEGES is optional and not mandatory in Mimer SQL.

The compliance of a certain statement is always compared to the current ANSI/ISO standard, which at the moment of writing is SQL-2016.

The table indicates how Mimer SQL complies to SQL-2016. Mimer SQL is fully compliant with all features in core SQL-2016, but Mimer SQL also supports a number of features outside core SQL-2016 which can be seen on the second row of the table.

Finally, extensions specific to Mimer SQL is described.

If portability over different database platforms is important, care should be taken to use standard SQL whenever possible. When you have to use Mimer SQL extensions these should be isolated so they can be exchanged when porting to another database. Even if you only use standard SQL there is no warranty that the code can be used with other database products as practically all vendors only implements a subset of the standard.

Chapter 3

Introduction to SQL Standards

The language SQL is standardized by international standard bodies such as ISO and ANSI. By using standard SQL it should be easier to move applications between different database systems without the need to rewrite a substantial amount of code. Using standard SQL does not give any warranty though as all vendors does not implement all features in the standard.

Mimer Information Technology's policy is to develop Mimer SQL as far as possible in accordance with the established standard. This enables users to switch to and from Mimer SQL easily.

The current standard for SQL is ISO/IEC 9075:2016, referred to here as SQL-2016.

The standard is written in a very formal manner. It is therefore difficult to use as a programming guide. SQL-2016 does not contain any specifications for administration of a database system and it does not specify any physical limitations. e.g. the maximum number of columns in table and maximum record size.

History of Standards

The first standard for SQL was published in 1986 and commonly known as SQL-1. An amendment to this standard, containing support for referential integrity, was published in 1989. The next major version of the standard was published in 1992 and is often referred to as SQL-92 (or SQL-2.) In 1996 a standard for stored procedures and functions was published (SQL/PSM), which was followed by the next major version, SQL-99. As mentioned, the current standard was published in 2016 and is called SQL-2016.

SQL-2016

This standard incorporates most of SQL-2016 and SQL/PSM. The publication of this standard made all earlier standards obsolete. SQL-2016 contains the following parts:

- Part 1: Framework (SQL/Framework)
- Part 2: Foundation (SQL/Foundation)
- Part 3: Call-Level Interface (SQL/CLI)
- Part 4: Persistent Stored Modules (SQL/PSM)
- Part 9: Management of External Data (SQL/MED)

- Part 10: Object Language Bindings (SQL/OLB)
- Part 11: Information and Definition Schema (SQL/Schemata)
- Part 13: Routines and Types Using the Java™ Programming Language (SQL/JRT)
- Part 14: XML-Related Specifications (SQL/XML)

Source: ISO/IEC 9075:2016(E) Information technology - Database languages - SQL.

The SQL-2016 standard contains different features and a subset of these features forms Core SQL-2016. The features included in Core SQL-2016 can be seen in *Chapter F, SQL-2016 Compliance*. This appendix also contains a list of non-core features supported by Mimer SQL.

The Unicode Standard and ISO/IEC 10646

The Unicode Standard is fully compatible with the international standard ISO/IEC 10646, Information Technology – Universal Multiple-Octet Coded Character Set (UCS).

While modeled on the ASCII character set, the Unicode Standard goes far beyond ASCII's limited ability to encode only the upper- and lowercase letters A through Z. It provides the capacity to encode all characters used for the written languages of the world – more than 1 million characters can be encoded. The Unicode character encoding treats alphabetic characters, ideographic characters, and symbols equivalently, which means they can be used in any mixture and with equal facility.

In addition to the Unicode standard, there is a technical standard called Unicode Collation Algorithm (UCA), which is kept synchronized with the ISO/IEC 14651 standard for International String Ordering.

EOR - European Ordering Rules

The Unicode Default Order and ISO/IEC 14651 have defined the default Latin alphabet to contain not only the base letters A through Z, but also a number of more or less language specific base letters. One example, the Romanian letter Î is not a variant of I; it is a separate base letter between I and J.

The EOR, European Ordering Rules, ENV 13710 (and ISO 12199 - Alphabetical ordering of multilingual terminological and lexicographical data represented in the Latin alphabet) have taken a more natural approach: The alphabet is A through Z, and the other language specific letters are secondary variants of the corresponding base letter.

Mimer SQL is using the EOR tailoring as the basis for all specific language tailorings.

Chapter 4

Mimer SQL Database Objects

This chapter provides a general introduction to the basic concepts of Mimer SQL databases and Mimer SQL objects.

Mimer SQL is a relational database system. This means that the information in the database is presented to the user in the form of tables. The tables represent a logical description of the contents of the database which is independent of, and insulates the user from, the physical storage format of the data.

The Mimer SQL database includes the data dictionary which is a set of tables describing the organization of the database and is used primarily by the database management system itself.

The database, although located on a single server, may be accessed from many distinct clients, linked over a network.

Commands are available for managing the connections to different databases, so the actual database being accessed may change during the course of an SQL session.

At any one time, however, the database may be regarded as one single organized collection of information.

System and Private Objects

Mimer SQL database objects can be divided into the following groups:

- **System Objects**

  System objects are global to the database. System object names must be unique for each object type since they are global and therefore common to all users.

  The system objects in a Mimer SQL database are: databanks, idents, schemas and shadows. A system object is owned by the ident that created it and only the creator of the object can drop it.

- **Private Objects**

  Private objects belong to a schema. Private object names are local to a schema, so two different schemas may contain an object with the same name. It is also possible to have objects with the same name in a schema, if they are of different types.

  The private objects in a Mimer SQL database are collations, domains, functions, indexes, modules, precompiled statements, procedures, sequences, synonyms, tables, triggers and views.

Private objects are usually fully identified by their qualified name, which is the name of the schema to which they belong and the name of the object in the following form: schema.object, see *Qualified Object Names* on page *39*.

Routines may exist in multiple versions having the same name. See *Mimer SQL Programmer's Manual, Chapter 11, Parameter Overloading*.

Conflicts arising from the use of the same object name in two different schemas are avoided when the qualified name is used. If a private object name is specified without explicit reference to its schema, it is assumed to belong to a schema with the same name as the current ident.

The Data Dictionary

The data dictionary contains information on all the database objects stored in a Mimer SQL database and how they relate to one another.

The data dictionary stores information about:

- Databanks, see *Databanks* on page *12*
- Idents, see *Idents* on page *14*
- Schemas, see *Schemas* on page *15*
- Tables and Views, see *Tables* on page *15*
- Indexes, see *Primary Keys and Indexes* on page *16*
- Functions and procedures, see *Routines – Functions and Procedures* on page *17*
- Modules, see *Modules* on page *18*
- Synonyms, see *Synonyms* on page *18*
- Triggers, see *Triggers* on page *19*
- Shadows, see *Shadows* on page *18*
- Sequences, see *Sequences* on page *19*
- Collations, see *Collations* on page *20*
- Domains, see *Domains* on page *22*
- Precompiled statements, see *Precompiled Statements* on page *20*
- Access rights and privileges, see *Privileges* on page *23*.

Databanks

A databank is the physical file where a collection of tables is stored. A Mimer SQL database can contain any number of databanks.

There are two types of databanks; system databanks and user databanks.

System Databanks

System databanks contain system information used by the database manager. These databanks are defined when the system is created.

The system databanks are:

- `SYSDB`, containing the data dictionary tables
- `TRANSDB`, used for transaction handling
- `LOGDB`, used for transaction logging
- `SQLDB`, used in transaction handling and for temporary storage of internal work tables.

User Databanks

User databanks contain the user tables. These databanks are defined by the user(s) responsible for setting up the database. See *Specifying the Location of User Databanks* on page *13* for details concerning path names for user databank files.

The division of tables between different user databanks is a physical file storage issue and does not affect the way the database contents are presented to the user. Except in special situations (such as when creating tables), databanks are completely invisible to the user.

Note: Backup and restore in Mimer SQL can be performed on a per-databank basis rather than on entire database basis. See the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data* for more information.

Specifying the Location of User Databanks

The location for a user databank file can be specified completely (as an absolute path name) or with some of the path name components omitted (a relative path name).

The default values used for omitted path name components are taken from the path name for the system databank file `SYSDB`, which is located in the database home directory.

Note: The databank location stored in the Mimer SQL data dictionary is the path name as explicitly specified, i.e. without the addition of default values for any omitted path name components. Such additions are determined and added each time the file is accessed.

Refer to the *Mimer SQL System Management Handbook, Chapter 2, The Database Environment* for recommendations concerning databank file management and for information on how the path name for a databank file is determined.

Idents

An ident is an authorization-id used to identify users, programs and groups. The different types of idents in a Mimer SQL database are USER, PROGRAM and GROUP idents.

USER Idents

USER idents identify individual users who can connect to a Mimer SQL database.

A USER ident's access to the database is usually protected by a password, and is also restricted by the specific privileges granted to the ident. USER idents are generally associated with specific physical individuals who are authorized to use the system.

For a USER ident it is possible to add one or several OS_USER logins which allows the user currently logged in to the operating system to access the Mimer SQL database without providing a password.

For example: if the current operating system user is ALBERT and there is an OS_USER login called ALBERT for an ident in Mimer SQL, ALBERT may start Mimer BSQL (for example) and connect directly to Mimer SQL simply by giving the ident name at the Username: prompt and press <return> at the password: prompt.

If the ident name is the same as the OS_USER login no ident name needs to be given, it is sufficient to press <return> at the username: prompt.

A USER ident may be defined without a password and in that case it is only possible to connect to Mimer SQL by using the OS_USER login. Dropping and adding password and OS_USER logins is done with ALTER IDENT statement.

PROGRAM Idents

PROGRAM idents do not strictly connect to Mimer SQL, but they may be entered from within an application program by using the ENTER statement.

The ENTER statement may only be used by an ident who is already connected to a Mimer SQL database.

An ident is granted the privilege to enter a PROGRAM ident. A PROGRAM ident is set up to have certain privileges and these apply after the ENTER statement has been used.

PROGRAM idents are generally associated with specific functions within the system, rather than with physical individuals.

The LEAVE statement is used to return to the state of privileges and database access that existed before ENTER was used.

GROUP Idents

GROUP idents are collective identities used to define groups of USER and/or PROGRAM idents.

Any privileges granted to or revoked from a GROUP ident automatically apply to all members of the group. Any ident can be a member of as many groups as required, and a group can include any number of members.

GROUP idents provide a facility for organizing the privilege structure in the database system. All idents are automatically members of a logical group which is specified in Mimer SQL statements by using the keyword PUBLIC.

Schemas

A schema defines a local environment within which private database objects can be created. The ident creating the schema has the right to create objects in it and to drop objects from it.

When a `USER` or `PROGRAM` ident is created, a schema with the same name is automatically created by default, and the created ident becomes the creator of the schema. This happens unless `WITHOUT SCHEMA` is specified in the `CREATE IDENT` statement. For idents who are not supposed to create database objects, it's good practice to specify `WITHOUT SCHEMA`.

When a private database object is created, the name for it can be specified in a fully qualified form which identifies the schema in which it is to be created. The names of objects must be unique within the schema to which they belong, according to the rules for the particular object-type.

If an unqualified name is specified for a private database object, a schema name equivalent to the name of the current ident is assumed.

Tables

Data in a relational database is logically organized in tables, which consist of horizontal rows and vertical columns. Columns are identified by a column-name. Each row in a table contains data pertaining to a specific entry in the database. Each field, defined by the intersection of a row and a column, contains a single item of data.

Each row in a table must have the same set of data items (one for each column in the table), but not all the items need to be filled in. A column can have a default value defined (either as part of the column specification itself or by using a domain with a default value) and this is stored in a field where an explicit value for the data item has not been specified.

If no default value has been defined for a column, the null value is stored when no data value is supplied (the way the null value is displayed depends on the application – in Mimer BSQL the minus sign is used).

A relational database is built up of several inter-dependent tables which can be joined together. Tables are joined by using related values that appear in one or more columns in each of the tables.

Part of the flexibility of a relational database structure is the ability to add more tables to an existing database. A new table can relate to an existing database structure by having columns with data that relates to the data in columns of the existing tables. No alterations to the existing data structure are required.

All data in a column contains information of one data type. The data type determines which data that can be stored in a column also the maximum length of the data. A data type may either be of fix or varying length. A fix data type will always use the same amount of physical space whereas a varying type only uses as much space as is needed. More information about data types can be found in *Data Types in SQL Statements* on page *44*.

Base Tables and Views

The logical representation of data in a Mimer SQL database is stored in tables. This is what the user sees, as distinct from the physical storage format which is transparent to the user.

The tables which store the data are referred to as base tables. Users can directly examine data in the base tables.

In addition, data may be presented in views, which are created from specific parts of one or more base tables. To the user, views may look the same as tables, but operations on views are actually performed on the underlying base tables.

Access privileges on views and their underlying base tables are completely independent of each other, so views provide a mechanism for setting up specific access to tables.

The essential difference between a table and a view is underlined by the action of the `DROP` command, which removes objects from the database. If a table is dropped, all data in the table is lost from the database and can only be recovered by redefining the table and re-entering the data. If a view is dropped, however, the table or tables on which the view is defined remain in the database, and no data is lost. Data may, however, become inaccessible to a user who was allowed to access the view but who is not permitted to access the underlying base table(s).

Note: Since views are logical representations of tables, all operations requested on a view are actually performed on the underlying base table, so care must be taken when granting access privileges on views.

Such privileges may include the right to insert, update and delete information. As an example, deleting a row from a view will remove the entire row from the underlying base table and this may include table columns the user of the view had no privilege to access.

Views may be created to simplify presentation of data to the user by including only some of the base table columns in the view or only by including selected rows from the base table. Views of this kind are called restriction views.

Views may also be created to combine information from several tables (join views). Join views can be used to present data in more natural or useful combinations than the base tables themselves provide (the optimal design of the base tables will have been governed by rules of relational database modeling). Join views may also contain restriction conditions.

Primary Keys and Indexes

Rows in a base table are uniquely identified by the value of the primary key defined for the table. The primary key for a table is composed of the values of one or more columns. A table cannot contain two rows with the same primary key value. (If the primary key contains more than one column, the key value is the combined value of all the columns in the key. Individual columns in the key may contain duplicate values as long as the whole key value is unique).

Other columns may also be defined as `UNIQUE`. A unique column is also a key, because it may not contain duplicate values, and need not necessarily be part of the primary key.

The columns of the primary key may not contain null values (this is one of the requirements of a strictly relational database).

Primary keys and unique columns are automatically indexed to facilitate effective information retrieval.

Other columns or combinations of columns may be defined as a secondary index to improve performance in data retrieval. Secondary indexes are defined on a table after it has been created (using the `CREATE INDEX` statement).

A secondary index may be useful when, for example, a search is regularly performed on a non-keyed column in a table with many rows, then defining an index on the column may speed up the search. The search result is not affected by the index but the speed of the search is optimized.

It should be noted, however, that indexes create an overhead for update, delete and insert operations because the index must also be updated.

An index can be used in select statements as an ordinary table, but explicit write operations on indexes are not allowed. There is no guarantee that the presence of an index will actually improve performance because the decision to use it or not is made by the internal query optimization process.

SQL queries are automatically optimized when they are internally prepared for execution. The optimization process determines the most effective way to execute the query and in some cases optimal query execution may not actually involve using an index.

Stored Procedures

In Mimer SQL you can define functions, procedures and modules, collectively known as stored procedures.

Mimer SQL stored procedures enable you to define and use powerful functionality through the creation and execution of routines. By using stored procedures, you can move application logic from the client to the server, thereby reducing network traffic.

Stored procedures are stored in the data dictionary and you can invoke them when needed.

For a complete and detailed discussion of stored procedures, see *Mimer SQL Programmer's Manual, Chapter 11, Mimer SQL Stored Procedures*.

Mimer SQL stored procedures are based on the ISO standard for Persistent Stored Modules (PSM).

Routines – Functions and Procedures

The term routine is a collective term for functions and procedures. Functions are distinguished from procedures in that they return a single value and the parameters of a function are used for input only. A function is invoked by using it where a value expression would normally be used.

Mimer SQL supports standard procedures and also result set procedures, which are procedures capable of returning the row value(s) of a result set.

Standard procedures are invoked directly by using the `CALL` statement and can pass values back to the calling environment through the procedure parameters.

In embedded SQL, result set procedures are invoked by declaring a cursor which includes the procedure call specification and by then using the `FETCH` statement to execute the procedure and return the row(s) of the result set.

In interactive SQL, a result set procedure is invoked by using the `CALL` statement directly and the result set values are presented in the same way as for a `SELECT` returning more than one row.

The creator of a routine must hold the appropriate access rights on any database objects referenced from within the routine. These access rights must remain as longs as the routine exists.

Routine names, like those of other private objects in the database, are qualified with the name of the schema to which they belong.

Modules

A module is simply a collection of routines. All the routines in a module are created when the module is created and belong to the same schema.

`EXECUTE` rights on the routines contained in a module are held on a per-routine basis, not on the module.

If a module is dropped, all the routines contained in the module are dropped.

Under certain circumstances a routine may be dropped because of the cascade effect of dropping some other database object. If such a routine is contained in a module, it is implicitly removed from the module and dropped. The other routines contained in the module remain unaffected.

In general, care should be taken when using `DROP` or `REVOKE` in connection with routines, modules or objects referenced from within routines because the cascade effects can often affect many other objects.

For more information, see *Mimer SQL User's Manual, Chapter 7, Dropping Objects from the Database*, and the *Mimer SQL User's Manual, Chapter 8, Revoking Privileges*.

Synonyms

A synonym is an alternative name for a table, view or another synonym. Synonyms can be created or dropped at any time.

Using synonyms can be a convenient way to address tables that are contained in another schema.

For example, if a view `customer_details` is contained in the schema `mimer_store`, the full name of the view is `mimer_store.customer_details`.

This view may be referenced from another schema `mimer_store_book` by its fully qualified name as given above. Alternatively, a synonym may be created for the view in schema `mimer_store_book`, e.g. `cust_details`. Then the name `cust_details` can simply be used to refer to the view.

Shadows

Mimer SQL Shadowing is a separate product you can use to create and maintain one or more copies of a databank on different disks. Shadowing provides extra protection from the consequences of disk crashes, etc. Shadowing requires a separate license.

Read more in the *Mimer SQL System Management Handbook, Chapter 10, Mimer SQL Shadowing*.

Triggers

A trigger defines a number of procedural SQL statements that are executed whenever a specified data manipulation statement is executed on the table or view on which the trigger has been created.

There are two types of triggers, row triggers and statement triggers. A row trigger is executed once for each row that is modified by a data manipulation operation. A statement trigger is invoked once for a data manipulation operation.

The trigger can be set up to execute `AFTER`, `BEFORE` or `INSTEAD OF` the data manipulation statement. Trigger execution can also be made conditional on a search condition specified as part of the trigger.

Read more in the *Mimer SQL Programmer's Manual, Chapter 12, Triggers*.

User-Defined Types and Methods

With user-defined types, it is possible to create new data types that can be used in table definitions and stored procedures.The data type used in a user-defined type definition may be a predefined data type or another user-defined type.

It is possible to define methods for a user-defined type. Methods are very similar to functions, they have only in parameters and return a single value. There are three different types of methods, constructor, instance and static methods.

Constructor methods are used to create new instances of a user-defined type. An instance method can only be used with an instance of a user-defined type. A static method is similar to a function, the only difference is how they are invoked. Both instance and constructor methods have an implicit parameter named `SELF` which represents an instance of a user-defined type.

It is possible to alter a user-defined type by adding or dropping methods.

An ident can use a user-defined type created by another ident, if the user has been granted usage privilege on the user-defined type. Likewise, in order to be able to use a method the user must have been granted execute privilege on the method.

Sequences

A sequence is a database object that provides a series of integer values.

A sequence has a start value, an increment step value and a minimum value and a maximum value defined when it is created (by using the `CREATE SEQUENCE` statement).

A sequence can be specified as having a certain data type which will determine the span of possible values for the sequence. The possible data types are `SMALLINT`, `INTEGER` and `BIGINT`.

A sequence with `CYCLE` option will generate its series of values repeatedly.

A sequence with `NO CYCLE` becomes exhausted when the end value has been used, and can not be used any more. (An exhausted sequence can be reset using the `ALTER SEQUENCE` statement.)

A sequence is created with an undefined value initially.

To generate the next value in the integer series of a sequence the `NEXT VALUE` function is used, see *NEXT VALUE* on page *111*. When this expression is used for the first time after the sequence has been created, it establishes the initial value for the sequence. Subsequent uses will establish the next value in the series of integer values of the sequence as the current value of the sequence.

It is also possible to get the current value of a sequence by using the `CURRENT VALUE` function, see *CURRENT VALUE* on page *100*. This function can not be used until the initial value has been established for the sequence (by using `NEXT VALUE` for the first time).

If a sequence is dropped with the `CASCADE` option in effect, column defaults referencing the sequence will be removed, but the columns will still exist. Similarly domain defaults referencing the sequence will be removed, but the domains will still exist. Other objects referencing the sequence will be dropped.

Precompiled Statements

A precompiled statement is a named query that can be executed by using this name. The query must be a DML statement, i.e. `DELETE`, `INSERT`, `SELECT` or `UPDATE`, or a `SET` or `CALL` statement. When the statement is created a compiled version of the query is stored in the data dictionary. Precompiled statements are primarily intended for use in mobile and embedded environments in which no SQL compiler is available due to limited memory resources.

Mimer SQL Character Sets

For character data, Mimer SQL uses the character set ISO 8859-1, also known as the Latin1 character set. By default, character data is sorted in the numerical order of its code according to the `ISO8BIT` collation.

For national character data, Mimer SQL uses the Unicode character set, which is a universal character set, see https://www.unicode.org for more information. National character data is sorted according to the `UCS_BASIC` collation. `UCS_BASIC` is a collation in which the ordering is determined entirely by the Unicode scalar values of the characters in the strings being sorted.

See the *Mimer SQL Reference Manual, Appendix B, Character Sets* for more information.

Collations

As stated in the previous section, character and national character data is sorted according to specific collations.

A collation, also known as a collating sequence, is a database object containing a set of rules that determines how character strings are compared, searched and alphabetically sorted. The rules in the collation determine whether one character string is less than, equal to or greater than another. A collation also determines how case-sensitivity and accents are handled.

You can specify a collation for ordering characters when you create or alter a table or create a domain.

If you have specified a collation for a column, the collation is used implicitly in SQL statements.

You only need to explicitly use a collate clause in SQL statements if you want to override the default collation or the collation you specified when creating or altering the column or creating the domain.

For more information, see *Mimer SQL User's Manual, Chapter 4, Collations*.

Data Integrity

A vital aspect of a Mimer SQL database is data integrity. Data integrity means that the data in the database is complete and consistent both at its creation and at all times during use.

Mimer SQL has built-in facilities to ensure the data integrity in the database:

- Primary keys and unique keys
- Foreign keys (also referred to as referential integrity)
- Domains
- Check constraints in table definitions
- Check options in view definitions

These features should be used whenever possible to protect the integrity of the database, guaranteeing that incorrect or inconsistent data is not entered into it. By applying data integrity constraints through the database management system, the responsibility of ensuring the data integrity of the database is moved from the users of the database to the database designer.

Primary Keys and Unique Keys

Rows in a base table are uniquely identified by the value of the primary key defined for the table. The primary key for a table is composed of the values of one or more columns. A table cannot contain two rows with the same primary key value. If the primary key contains more than one column, the key value is the combined value of all the columns in the key. Individual columns in the key may contain duplicate values as long as the whole key value is unique.

Apart from a primary key constraint its also possible to add one or more unique constraints. The primary key constraint and the unique constraint are similar, but treat null values in different ways. A null value can never be stored in a primary key column, but a unique constraint column can contain null values.

The definition of the primary key is also a definition of the most effective access path for the table.

Foreign Keys – Referential Integrity

A foreign key is one or more columns in a table defined as cross-referencing the primary key or a unique key of a table.

Data entered into the foreign key must either exist in the key that it cross-references or be null. This maintains referential integrity in the database, ensuring that a table can only contain data that already exists in the selected key of the referenced table.

As a consequence of this, a key value that is cross-referenced by a foreign key of another table must not be removed from the table to which it belongs by an update or delete operation if this ultimately violates the referential constraint.

The DELETE rule defined for the referential constraint provides a mechanism for adjusting the values in a foreign key in a way that may permit a cross-referenced key value to effectively be removed.

Similarly, the UPDATE rule defined for the referential constraint provides a mechanism for adjusting the values in a foreign key in a way that may permit a cross-referenced key value to effectively be updated.

Note: The referential integrity constraints are effectively checked at the end of an INSERT, DELETE or UPDATE statement, or at COMMIT depending on whether the constraint is declared as IMMEDIATE or DEFERRED.

Foreign key relationships are defined when a table is created using the CREATE TABLE statement and can be added to an existing table by using the ALTER TABLE statement.

The cross-referenced table must exist prior to the declaration of foreign keys on that table, unless the cross-referenced and referencing tables are the same.

If foreign key relationships are defined for tables in a CREATE SCHEMA statement, it is possible to reference a table that will not be created until later in the CREATE SCHEMA statement.

Note: Both the table containing the foreign key and the cross-referenced table must be stored in a databank with either the TRANSACTION or LOG option.

Domains

Each column in a table holds data of a single data type and length, specified when the column is created or altered. The data type may be specified explicitly, e.g. CHARACTER(20) or INTEGER, or through the use of domains, which can give more precise control over the data that will be accepted in the column.

A domain definition consists of a data type with optional check conditions and an optional default value. Data which falls outside the constraints defined by the check conditions is not accepted in a column which is defined using the domain. If a variable or parameter in a stored routine is defined as a domain with a check constraint, it is not possible to assign the parameter or variable a value which is allowed by the check constraint.

A column defined using a domain for which a default value is defined will automatically receive that value if row data is entered without a value being explicitly specified.

A variable in a stored routine or trigger declared using a domain for which a default value is defined will automatically receive that value unless an explicit default clause is present in the declaration.

In order for an ident to create a table containing columns whose data type is defined through the use of a domain, the ident must first have been granted USAGE rights on it, see the *Mimer SQL User's Manual, Chapter 8, Granting Privileges*.

Check Constraints

Check constraints may be specified in table and domain definitions to make sure that the values in a table row conform to certain conditions. See the *Mimer SQL User's Manual, Chapter 7, Check Constraints* for more information.

Check Options in View Definitions

You can maintain view integrity by including a check option in the view definition. This causes data entered through the view to be checked against the view definition. If the data conflicts with the conditions in the view definition, it is rejected.

Privileges

Privileges control users' access to database objects and the operations they can perform in the database.

`USER` and `PROGRAM` idents are usually protected by a password, which must be given together with the correct ident name in order for a user to gain access to the database or to enter a program ident. (Alternatively, an OS_USER login can be used to login without providing a password.)

Passwords are stored in encrypted form in the data dictionary and cannot be read by any ident, including the system administrator. A password may only be changed by the ident to which it belongs or by the creator of the ident.

A set of access and privileges define the operations each ident is permitted to perform. There are three classes of privileges in a Mimer SQL database:

- system privileges
- object privileges
- access privileges.

System Privileges

System privileges control the right to perform backup and restore operations, the right to execute the `UPDATE STATISTICS` statement as well as the right to create new databanks, idents, schemas and to manage shadows.

System privileges are granted to the system administrator when the system is installed and may be granted by the administrator to other idents in the database. As a general rule, system privileges should be granted to a restricted group of users.

Note: An ident who is given the privilege to create new idents is also able to create new schemas.

Object Privileges

Object privileges control membership in group idents, the right to invoke functions and procedures, the right to enter program idents, the right to create new tables or sequences in a specified databank and the right to use a domain or sequence.

The creator of an object is automatically granted full privileges on that object.

Thus the creator of:

- a group is automatically a member of the group
- a function or procedure may execute the routine
- a program ident may enter the program ident
- a schema may create objects in and drop objects from the schema
- a databank may create tables and sequences in the databank

- a table has all access rights on the table
- a domain may use that domain
- a sequence may use that sequence.

The creator of an object generally has the right to grant any of these privileges to other users. In the case of functions and procedures, this actually depends on the creator's access rights on objects referenced from within the routine.

Access Privileges

Access privileges define access to the contents of the database, i.e. the rights to retrieve data from tables or views, delete data, insert new rows, update data and to refer to table columns as foreign key references.

About Privileges

Granted privileges can be regarded as instances of grantor/privilege stored for an ident. An ident will hold more than one instance of a privilege if different grantors grant it.

A privilege will be held as long as at least one instance of that privilege is stored for the ident. All privileges may be granted with the `WITH GRANT OPTION` which means that the receiver has, in turn, the right to grant the privilege to other idents. An ident will hold a privilege with the `WITH GRANT OPTION` as long as at least one of the instances stored for the ident was granted with this option.

If the same grantor grants a privilege to an ident more than once, this will not result in more than one instance of the privilege being recorded for the ident. If a particular grantor grants a privilege without the `WITH GRANT OPTION` and subsequently grants the privilege again with the `WITH GRANT OPTION`, the `WITH GRANT OPTION` will be added to the existing instance of the privilege.

Each instance of a privilege held by an ident is revoked separately by the appropriate grantor. It is possible to revoke the `WITH GRANT OPTION` without revoking the associated privilege completely. See the *Mimer SQL User's Manual, Chapter 8, Defining Privileges* for more information.

Chapter 5

Collations and Linguistic Sorting

This chapter provides the basic concepts of national characters and linguistic sorting.

The default Unicode sorting order is provided in https://www.unicode.org/Public/UCA/latest/allkeys.txt.

This table (the Default Unicode Collation Element Table) provides a mapping from characters to collation elements for all the explicitly weighted characters.

Multilevel Comparisons

There are different levels of comparisons to pay attention to, such as case and accent sensitivity. From the Default Unicode Collation Element Table, referred to above, the following definition for the letter b is picked:

```
0062  ; [.0A29.0020.0002.0062] # LATIN SMALL LETTER B
```

Within square brackets there are four levels of comparison keys; the Primary level, the Secondary level, the Tertiary level and the Quaternary level.

Primary level:

Typically, this is used to denote differences between base characters (for example, `a` < `b`). It is the strongest difference. For example, dictionaries are divided into different sections by the base character. This is also called the level-1 strength.

Mimer SQL's predefined level 1 collations have names ending with `_1`, e.g. `ENGLISH_1`.

Secondary level:

Accents in the characters are usually considered secondary differences (for example, `ab` < `áb` < `ac`). A secondary difference is ignored when there is a primary difference anywhere in the strings. This is also called the level-2 strength.

Mimer SQL's predefined level 2 collations have names ending with `_2`, e.g. `ENGLISH_2`.

Note: In some languages (such as Icelandic), certain accented letters are considered to be separate base characters.

Tertiary level:

Upper and lower case differences in characters are distinguished at the tertiary level (for example, ab < Ab < áb). In addition, a variant of a letter differs from the base form on the tertiary level (such as a and ª). A tertiary difference is ignored when there is a primary or secondary difference anywhere in the strings. This is also called level-3 strength.

Mimer SQL's predefined level 3 collations have names ending with _3, e.g. ENGLISH_3.

Quaternary level:

When punctuation is ignored (such as space and hyphen) at level 1-3, an additional level can be used to distinguish words with and without punctuation (for example, ab < a c < a-c < ac). A quaternary difference is ignored when there is a primary, secondary, or tertiary difference. This is also called the level-4 strength.

Multilevel comparison means the following: Two strings are compared on the primary level. If the comparison for this level fails to establish a unique and determined sequence for the strings, the second level are taken into consideration. If this likewise fails to produce a unique sequence, the tertiary level is invoked, and after this the quaternary level is used. If still a unique sequence can not be established, the two strings are regarded as equivalent.

How far to go in this comparison chain is decided by the definition of the collation used on the data. See *Tailorings* on page *27*.

Alternate Weighting

Alternate collation elements, i.e. punctuation, can be treated different depending on the weighting method used:

Non-ignorable

Alternate collation elements are treated as normal collation elements. This is the default option.

Shifted

Alternate collation elements are set to zero at the primary, secondary and tertiary level, and the fourth-level weight is set to the primary weight. All other collation elements, with a non-zero primary weight, will receive a fourth-level weight of 0xFFFF. If the primary weight is zero, the fourth-level weight is also zero.

Shift-trimmed

Alternate collation elements are set to zero at the primary, secondary and tertiary level, and the fourth-level weight is set to the primary weight. All other collation elements are set to zero. This will emulate POSIX behavior.

The following gives an example of the alternate weighting differences.

Non-ignorable	Shifted	Shift-trimmed
de luge	death	death
de Luge	de luge	deluge
de-luge	de-luge	de luge
de-Luge	deluge	de-luge
death	de Luge	deLuge
deluge	de-Luge	de Luge
deLuge	deLuge	de-Luge
demark	demark	demark

Tailorings

A tailoring is a set of rules and attributes that forms a so called collation delta string, which is used as the basis when creating a collation. When a new collation is to be created, the tailoring describes how to modify an existing collation definition to get the new one. A collation is created by the `CREATE COLLATION` statement, see *CREATE COLLATION* on page *251*.

Attributes

When creating a collation, the tailoring string can include attribute settings for comparison level, accent order, which case that should be first in order and alternate weighting.

Attributes are optional.

Option	Values	Description

Level	`[Level 1]`
	`[Level 2]`
	`[Level 3]`
	`[Level 4]`
		Sort level for the collation.
		`[Level 3]` is default.

Accent order	`[AccentOrder Forward]` `[AccentOrder Backward]`	Secondary level ordering direction. `[AccentOrder Forward]` is default.
Case first	`[CaseFirst Lower]` `[CaseFirst Upper]`	Tertiary level case ordering. `[CaseFirst Lower]` is default.

Alternate	`[Alternate Non-ignorable]`
	`[Alternate Shifted]`
	`[Alternate Shift-trimmed]`
		Alternatives for punctuation.
		`[Alternate Non-ignorable]` is default.

Hiragana	`[Hiragana On]` `[Hiragana Off]`	Option for Japanese sorting. Use `[Hiragana On][Level 4]` for full Japans ordering. `[Hiragana Off]` is default.
Numeric	`[Numeric On]` `[Numeric Off]`	Option for numeric sorting. `[Numeric Off]` is default.

Special sort rules

Language	Attribute	Description

Chinese	`[CJK KangXi]`
	`[CJK PinYin]`
	`[CJK Stroke]`
	`[CJK ZhuYin]`
		Use special sort rules for Chinese characters.

Japanese	`[CJK Kanji]`	Use the JIS X 4061-1996 collation rules. Gives proper ordering of PROLONGED SOUND MARK and ITERATION MARK.
Korean	`[CJK Hanja]`	Sort Hanja characters secondary different from the corresponding Hangul character.
Vietnamese	`[CJK ChuNom]`	Use syllable by syllable processing. In lexical ordering, differences in letters are treated as primary, differences in tone markings as secondary, and differences in case as tertiary differences. Ordering according to primary and secondary differences proceeds syllable by syllable. According to this principle, a dictionary lists “ban mai” before “bàn cát” because the secondary difference in the first syllable takes precedence over the primary difference in the second.
Assamese, Bengali, Gujarati, Hindi, Kannada, Konkani, Malayalam, Manipuri, Marathi, Nepali, Oriya, Punjabi, Sanskrit, Tamil, Telugu	`[Indic]`	Use traditional collation rules for Indic languages, which provides for proper sorting of words ending with a dead consonant (without an inherent vowel).

Rules

The rules in a tailoring string defines how to change the underlying collation. Each rule contains a string of ordered characters that starts with a reset value.

Symbol	Example	Example description
`&`	`&Z`	Reset at this letter. Rules will be relative to this letter from here on.
`<`	`a < b`	Identifies a primary level difference between `a` and `b`.
`<<`	`e << ê`	Identifies a secondary level difference between `e` and `ê`.
`<<<`	`s <<< S`	Identifies a tertiary level difference between `s` and `S`.
`=`	`i = y`	Signifies no difference between `i` and `y`.
`"`	`","`	The quoted character `,` (comma).
`#`	`#0141#`	Hexadecimal representation of Polish L with stroke.

Note: `;` can be used to represent secondary relations and `,` to represent tertiary relations, instead of `<<` and `<<<` respectively.

Example

The following is a Danish tailoring example:

```
[level 4]
[casefirst upper]
[alternate shifted]
& Y << ü <<< Ü
& Z < æ <<< Æ << ä <<< Ä < ø <<< Ø << ö <<< Ö < å <<< Å << aa
<<< Aa <<< AA
```

Sorting Examples

Numerical data sorting

Here is an example on how to sort numerical data properly:

```
SQL>CREATE TABLE alphanum (codes VARCHAR(10));
SQL>INSERT INTO  alphanum VALUES('A123');
SQL>INSERT INTO  alphanum VALUES('A234');
SQL>INSERT INTO  alphanum VALUES('A23');
SQL>INSERT INTO  alphanum VALUES('A3');
SQL>INSERT INTO  alphanum VALUES('A1');

SQL>-- Regular order [Numeric Off]
SQL>SELECT * FROM alphanum ORDER BY codes;

CODES
==========
A1
A123
A23
A234
A3

SQL>-- Numeric order [Numeric On]
SQL>CREATE COLLATION numeric FROM eor USING '[Numeric On]';
SQL>SELECT * FROM alphanum ORDER BY codes COLLATE numeric;

CODES
==========
A1
A3
A23
A123
A234
```

Two column sorting

Here is an example on how to sort two fields properly; in this case 'last name', 'first name':

```
SQL>create table name(last varchar(32),first varchar(32));
SQL>insert into  name values('van Diesel','Peter');
SQL>insert into  name values('van Diesel','Thomas');
SQL>insert into  name values('vanDiesel','Peter');
SQL>insert into  name values('vanDiesel','Thomas');
SQL>insert into  name values('Van Diesel','Peter');
SQL>insert into  name values('Van Diesel','Thomas');
SQL>insert into  name values('Van','Stephan');
SQL>insert into  name values('Van','Buster');
SQL>create collation names from EOR_1
SQL&using '[level 4][alternate shifted]&9<","'; -- ',' before 'A'
SQL>select last || ', ' || first as fullname
SQL&from name order by fullname collate names;

FULLNAME
========
Van, Buster
Van, Stephan
van Diesel, Peter
vanDiesel, Peter
Van Diesel, Peter
van Diesel, Thomas
vanDiesel, Thomas
Van Diesel, Thomas
```

Name prefix handling

Example on how to treat different Mac prefixes as equal. Typical names are MacAlister, McAlister, McDonell, MacDougel and M'Dougel.

```
SQL>create collation mac_english_3 from english_3 using
SQL&'&MAC<<<mc<<<Mc<<<MC<<<m#27#<<<M#27#';
SQL>create collation mac_english_2 from mac_english_3 using '[level 2]';
SQL>create table macs (name varchar(32));
SQL>insert into macs values('M''Dougel');
SQL>insert into macs values('McDonell');
SQL>insert into macs values('MacAlister');
SQL>insert into macs values('McAlister');
SQL>insert into macs values('MacDougel');
SQL>select * from macs order by name collate english_3;
name
================================
M'Dougel
MacAlister
MacDougel
McAlister
McDonell

                5 rows found

SQL>select * from macs order by name collate mac_english_3;
name
================================
MacAlister
McAlister
McDonell
MacDougel
M'Dougel

                5 rows found

SQL>select * from macs where name = 'macalister' collate mac_english_2;
name
================================
MacAlister
McAlister

                2 rows found

SQL>select * from macs where name = 'mcalister' collate mac_english_2;
name
================================
MacAlister
McAlister

                2 rows found

SQL>select * from macs where name = 'm''alister' collate mac_english_2;
name
================================
MacAlister
McAlister

                2 rows found
```

Collating Details

Expanding Characters

A single character can map to a sequence of collation elements. For instance, ß is equivalent to ss. In German Phonebook ä, ö and ü sort as though they were ae, oe and ue respectively.

Contracting Character Sequences

Many languages have digraphs, which actually counts as separate letters. In traditional Spanish, ch sorts between c and d, and ll sorts between l and m. Two characters are mapped into a single collation element that cause the combination to be ordered differently from either character individually.

Another example of contractions are lj and nj in Bosnian and Croatian, which sorts after l and n respectively.

Order without contraction	Order with contraction “nj” sorting after “n”
Na	Na
Ni	Ni
Nj	Nk
Nja	Nz
Njz	Nj
Nk	Nja
Nz	Njz
Oa	Oa

Backward Accent Ordering

Some languages, particularly French, require words to be ordered on the secondary level by comparing backwards from right to left.

Example

English ordering	French ordering
Cote	Cote
Coté	Côte
Côte	Coté
Côté	Côté

Indic

Attribute: [indic]
Function: Method for traditional Indic collation

The traditional Indic sort order is as follows:

1. Vowel
2. Vowelless consonant
3. Vowelless consonant + Vowel
4. Vowelless consonant + Vowelless consonant
5. Vowelless consonant + Vowelless consonant + Vowel
6. ... and so on

As the consonant letters in Indic scripts includes an inherent vowel /a/, the following transformations are applied before sorting:

1. Consonant + Virama => Vowelless consonant
2. Consonant + Vowel-sign => Vowelless consonant + Vowel
3. Consonant => Vowelless consonant + A

Transformation examples:

क्	ka + virama	=> क्	k
क	ka	=> क्अ	k + A
कि	ka + i	=> क्इ	k + I
कु	ka + u	=> क्उ	k + U
के	ka + e	=> क्ए	k + E
को	ka + o	=> क्ओ	k + O
क्क	ka + virama + ka	=> क्क्अ	k + k + A

The method for traditional Indic collation effectively works for the following scripts:

- Devanagari (Hindi, Konkani, Marathi, Nepali and Sanskrit)
- Bengali (Assamese, Bengali and Manipuri)
- Gujarati
- Oriya
- Telugu
- Kannada
- Malayalam

The famous authoritative Monier-Williams: Sanskrit-English Dictionary is a good reference:

https://www.ibiblio.org/sripedia/ebooks/mw/

https://www.ibiblio.org/sripedia/ebooks/mw/0000/mw__0033.html

The `[indic]` attribute also works for Tamil, but with different rules as used in the authoritative University of Madras: Tamil Lexicon http://dsal.uchicago.edu/dictionaries/tamil-lex/

Punjabi does not need any tailoring, the default order follows the rules in the Punjabi University: Punjabi-English Dictionary ISBN:8173800960.

Without the [indic] attribute, a very large tailoring is needed for traditional collation. See the following example for Devanagari.

```
&#0903#<#0915##094D#
&#0915#<#0916##094D#
&#0916#<#0917##094D#
&#0917#<#0918##094D#
&#0918#<#0919##094D#
&#0919#<#091A##094D#
&#091A#<#091B##094D#
&#091B#<#091C##094D#
&#091C#<#091D##094D#
&#091D#<#091E##094D#
&#091E#<#091F##094D#
&#091F#<#0920##094D#
&#0920#<#0921##094D#
&#0921#<#0922##094D#
&#0922#<#0923##094D#
&#0923#<#0924##094D#
&#0924#<#0925##094D#
&#0925#<#0926##094D#
&#0926#<#0927##094D#
&#0927#<#0928##094D#
&#0928#<#092A##094D#
&#092A#<#092B##094D#
&#092B#<#092C##094D#
&#092C#<#092D##094D#
&#092D#<#092E##094D#
&#092E#<#092F##094D#
&#092F#<#0930##094D#
&#0930#<#0932##094D#
&#0932#<#0933##094D#
&#0933#<#0935##094D#
&#0935#<#0936##094D#
&#0936#<#0937##094D#
&#0937#<#0938##094D#
&#0938#<#0939##094D#

&#0915##094D##0905#=#0915#
&#0915##094D##0906#=#0915##093E#
&#0915##094D##0907#=#0915##093F#
&#0915##094D##0908#=#0915##0940#
&#0915##094D##0909#=#0915##0941#
&#0915##094D##090A#=#0915##0942#
&#0915##094D##090B#=#0915##0943#
&#0915##094D##0960#=#0915##0944#
&#0915##094D##090C#=#0915##0962#
&#0915##094D##0961#=#0915##0963#
&#0915##094D##090D#=#0915##0945#
&#0915##094D##090E#=#0915##0946#
&#0915##094D##090F#=#0915##0947#
&#0915##094D##0910#=#0915##0948#
&#0915##094D##0911#=#0915##0949#
&#0915##094D##0912#=#0915##094A#
&#0915##094D##0913#=#0915##094B#
&#0915##094D##0914#=#0915##094C#

...
same pattern for #0916#..#0938# (32)
...

&#0939##094D##0905#=#0939#
&#0939##094D##0906#=#0939##093E#
&#0939##094D##0907#=#0939##093F#
&#0939##094D##0908#=#0939##0940#
&#0939##094D##0909#=#0939##0941#
&#0939##094D##090A#=#0939##0942#
&#0939##094D##090B#=#0939##0943#
```

```
&#0939##094D##0960#=#0939##0944#
&#0939##094D##090C#=#0939##0962#
&#0939##094D##0961#=#0939##0963#
&#0939##094D##090D#=#0939##0945#
&#0939##094D##090E#=#0939##0946#
&#0939##094D##090F#=#0939##0947#
&#0939##094D##0910#=#0939##0948#
&#0939##094D##0911#=#0939##0949#
&#0939##094D##0912#=#0939##094A#
&#0939##094D##0913#=#0939##094B#
&#0939##094D##0914#=#0939##094C#
```

Japanese

Attribute: [CJK Kanji]
Function: JIS X 4061-1996 rules for SOUND/ITERATION MARKS

This attribute is an implementation of JIS X 4061-1996 and the collation rules are based on that standard.

The following criteria are considered in order until the collation order is determined. By default, Levels 1 to 4 are applied and Level 5 is ignored (as JIS does).

Level 1: alphabetic ordering

The character classes are sorted in the following order:

```
Space characters, Symbols and Punctuations, Digits,
Latin Letters, Greek Letters, Cyrillic Letters,
Hiragana/Katakana letters, Kanji ideographs.
```

In the class, alphabets are collated alphabetically; Kana letters are AIUEO-betically (in the Gozyuon order).

For Kanji, see *Kanji Classes on page 36*.

Other characters are collated as defined.

Characters not defined as a collation element are ignored and skipped on collation.

Level 2: diacritic ordering

In the Latin vowels, the order is as shown the following list.

```
One without diacritical mark, then with diacritical mark.
```

In Kana, the order is as shown the following list.

```
A voiceless kana, the voiced, then the semi-voiced
(if exists). (eg. Ka before Ga; Ha before Ba before Pa)
```

Level 3: case ordering

A small Latin character is less than the corresponding capital character.

In Kana, the order is as shown in the following list:

```
replaced PROLONGED SOUND MARK(U+30FC);
Small Kana;
replaced ITERATION MARK (U+309D, U+309E, U+30FD or U+30FE);
then normal kana
```

For example, `Katakana A + PROLONGED SOUND MARK, Katakana A + Small Katakana A, Katakana A + ITERATION MARK, Katakana A + Katakana A`.

Level 4: variant ordering

Hiragana is lesser than Katakana.

Level 5: width ordering

A character that belongs to the block `Halfwidth and Fullwidth Forms` is greater than the corresponding normal character.

Note: According to the JIS standard, the level 5 should be ignored.

Kanji Classes

There are three Kanji classes:

1 **The 'saisho' (minimum) Kanji class**

It comprises five Kanji-like characters, i.e. U+3003, U+3005, U+4EDD, U+3006, U+3007. Any Kanji except U+4EDD are ignored on collation.

2 **The 'kihon' (basic) Kanji class**

It comprises JIS levels 1 and 2 kanji in addition to the minimum Kanji class. Sorted in the JIS order. Any Kanji excepting those defined by JIS X 0208 are ignored on collation.

3 **The 'kakucho' (extended) Kanji class**

All the CJK Unified Ideographs in addition to the minimum Kanji class. Sorted in the Unicode order.

Note: This is the implemented class.

Korean

Attribute: [CJK Hanja]
Function: Special sort table access

Hanja characters are sorted with secondary difference from the corresponding Hangul character.

Vietnamese

Attribute: [CJK ChuNom]
Function: Syllable by syllable processing

In lexical ordering, differences in letters are treated as primary, differences in tone markings as secondary, and differences in case as tertiary differences. Ordering according to primary and secondary differences proceeds syllable by syllable. According to this principle, a dictionary lists “ban mai” before “bàn cát” because the secondary difference in the first syllable takes precedence over the primary difference in the second.

Chapter 6

SQL Syntax Elements

This chapter presents the basic elements of the SQL language and the simplest relationships between them.

It covers the following units:

Syntax unit	Summary	Section
Separators	Syntax element delimiters.	*Separators* on page *37*
Special characters	Syntax pattern characters.	*Special Characters* on page *37*
Identifiers	SQL identifiers, host variable names and keywords.	*Identifiers* on page *38*
Data types	Character, integer, decimal, floating point, date, time, timestamp, interval, binary, boolean.	*Data Types in SQL Statements* on page *44*
Literals	Character, integer, decimal, floating point, date, time, timestamp, interval, binary, boolean.	*Literals* on page *64*

Separators

Characters having the Unicode property “White_Space” are used as separators, e.g. <TAB>, <LF>, <VT>, <FF>, <CR> and <SP>.

Special Characters

Certain special characters have particular meanings in SQL statements; for example: delimiters, double quotation marks, single quotation marks and arithmetic and comparative operators.

The special characters `$` and `#` may, in some circumstances, be used in the same contexts as letters, see *Identifiers* on page *38*.

A separator is used to separate keywords, identifiers and literals from each other.

Identifiers

An identifier is defined as a sequence of one or more characters forming a unique name.

Identifiers are constructed according to certain fixed rules. It is useful to distinguish between SQL identifiers, which are local to SQL statements and host identifiers, which relate to the host programming language.

Rules for constructing host identifiers may vary between host languages.

SQL Identifiers

SQL identifiers consist of a sequence of one or more Unicode characters. The maximum length of an SQL identifier is 128 characters.

SQL identifiers (except for delimited identifiers) must begin with a character having the Unicode property "ID_Start" or one of the special characters `$` or `#`, and may then contain characters having the Unicode property "ID_Continue". For a detailed description, see https://www.unicode.org/reports/tr31.

The case of letters in SQL identifiers is not significant, not even if it is a delimited identifier.

Delimited Identifiers

Delimited identifiers means identifiers enclosed in double quotation marks: `""`. Such identifiers are special in two aspects:

- They can contain characters normally not supported in SQL identifiers.
- They can be identical to a reserved word.

Two consecutive double quotation marks within a delimited identifier are interpreted as one double quotation mark.

Unicode Delimited Identifiers

A Unicode delimited identifier consists of a sequence of Unicode characters enclosed in double quotation marks and preceded by the letter `U` and an ampersand, i.e. `U&`. Unicode characters can be given by four hexadecimal digits preceded by a backslash character (`\`), or by six hexadecimal digits preceded with a backslash character and a plus character (`\+`).

Two consecutive backslash characters within a Unicode delimited identifier are interpreted as a single backslash character.

A Unicode delimited identifier is typically used when an identifier contains a character difficult to type using the keyboard. For example the identifier `München` can be given as `U&"M\00FCnchen"`.

Examples

The following examples illustrate the general rules for forming SQL identifiers:

Valid	Invalid	Explanation
`COLUMN_1`	`COLUMN+1`	COLUMN+1 is an expression
`#14`	`14`	14 is an integer literal
`"MODULE"`	`MODULE`	MODULE is a reserved word

Valid	Invalid	Explanation
`U&"M\00FCnchen"`		Unicode delimited identifier for München.

Note: Leading blanks are significant in delimited identifiers.

Naming Objects

Objects in the database may be divided into two classes:

System Objects

System objects, such as databanks, idents, schemas and shadows, are global to the system. System object names must be unique within each object class since they are common to all users. System objects are uniquely identified by their name alone.

Private Objects

Private objects, such as domains, functions, indexes, modules, precompiled statements, procedures, sequences, synonyms, tables, triggers and views, belong to a schema and have names that are local to that schema. In a given schema, the names used for tables, synonyms, views, indexes and constraints must be unique within that group of objects, i.e. a table cannot have a name that is already being used by a synonym, view, index or constraint etc. Similarly, in a given schema, the names used for domains must be unique within that group of objects.

Functions and procedures may have the same name as long as they differ with regard to the number of parameters or the data type of the parameter. See *Mimer SQL Programmer's Manual, Chapter 11, Parameter Overloading*.

The names of all other objects, modules and sequences in the schema must be unique within their respective object-type. Two different schemas may contain objects of the same type with the same name. Private objects are uniquely identified by their qualified name (see below).

Qualified Object Names

Names of private objects in the database may always be qualified by the name of the schema to which they belong. The schema name is separated from the object name by a period, with the general syntax: `schema.object`.

If a qualified object name is specified when an object is created, it will be created in the named schema. If an object name is unqualified, a schema name with the same name as the current ident is assumed.

It is recommended that object names are always qualified with the schema name in SQL statements, to avoid confusion if the same program is run by different Mimer SQL idents.

When the name of a column is expressed in its unqualified form it is syntactically referred to as a `column-name`.

When the name of a column must be expressed unambiguously it is generally expressed in its fully qualified form, i.e. `schema.table.column` or `table.column`, and this is syntactically referred to as a `column-reference`.

It is possible for a `column-reference` to be the unqualified name of a column in contexts where this is sufficient to unambiguously identify the column.

When the name of a column is used to indicate the column itself, e.g. in `CREATE TABLE` statements, a `column-name` must be used, i.e. the name of the column cannot be qualified.

The exception to this is in the `COMMENT ON COLUMN` statement where a `column-reference` is required because the name of the column must be qualified by the name of the table or view to which it belongs.

The contexts where the name of a column refers to the values stored in the column are:

- in expressions
- in set functions
- in search conditions
- in `GROUP BY` clauses.

In these contexts a `column-reference` must be used to identify the column.

The column name qualifiers which may be used in a particular SQL statement are determined by the way the table is identified in the `FROM` clause of the `SELECT` statement.

Alternative names (correlation names) may be introduced in the `FROM` clause, and the table reference used to qualify column names must conform to the following rules:

- **If no correlation names are introduced:**

  The column name qualifier is the table name exactly as it appears in the `FROM` clause.

  For example:

  ```
  SELECT BOOKADM.HOTEL.NAME, ROOMS.ROOMNO
  FROM BOOKADM.HOTEL JOIN ROOMS ...
  ```

  but not

  ```
  SELECT BOOKADM.HOTEL.NAME, ROOMS.ROOMNO
  FROM HOTEL JOIN BOOKADM.ROOMS ...
  ```

- **If a correlation name is introduced:**

  The correlation name and not the original table reference, may be used to qualify a column name. The correlation name may not itself be qualified.

  For example:

  ```
  SELECT H.NAME, ROOMS.ROOMNO
  FROM HOTEL H JOIN ROOMS ...
  ```

  but not

  ```
  SELECT HOTEL.NAME, ROOMS.ROOMNO
  FROM HOTEL H JOIN ROOMS ...
  ```

Outer References

In some constructions where subqueries are used in search conditions, see *Chapter 11, The SELECT Expression*, it may be necessary to refer in the lower level subquery to a value in the current row of a table addressed at the higher level.

A reference to a column of a table identified at a higher level is called an outer reference. The following example shows the outer reference in bold type:

```
SELECT NAME
FROM HOTEL
WHERE EXISTS (SELECT * FROM
              FROM  BOOK_GUEST
              WHERE HOTELCODE = HOTEL.HOTELCODE)
```

The lower-level subquery is evaluated for every row in the higher level result table. The example selects the name of every hotel with at least one entry in the BOOK_GUEST table.

A qualified column name is an outer reference if, and only if, the following conditions are met:

- The qualified column name is used in a search condition of a subquery.
- The qualifying name is not introduced in the `FROM` clause of that subquery.
- The qualifying name is introduced at some higher level.
- The qualified column name is valid everywhere in a subquery.

Parameter Markers and Host Identifiers

Parameter markers and host identifiers are used when passing input or output data. The concepts are very similar, the major difference is that parameter markers are used in dynamic SQL, where the parameter marker data type is decided at `PREPARE` time, while a host identifier is declared and has a defined data type.

Parameter Markers

A parameter marker is put in the location of an input or output expression in a prepared SQL statement.

Parameter markers are assigned data types appropriate to their usage. See the *Mimer SQL Programmer's Manual, Chapter 4, Dynamic SQL,* for a discussion of dynamic SQL. For parameter markers used to represent data assigned to columns, the data type is in accordance with the column definition.

Mimer SQL supports different styles of parameter markers:

- **Question mark parameter marker**

  A question mark parameter marker (`?`) will be NOT NULL or NULL depending on the input or output expression.

- **Colon notation parameter marker**

  A colon notation parameter marker is specified as a colon followed by a parameter name, e.g. `:lastname`. A null indicator should be provided if the input or output expression is nullable, e.g. `:lastname:indic`.

  If the same parameter marker name is used several times in an SQL statement, it is considered to be one parameter marker. A parameter marker name can only be used several times if the implied data types of the parameter markers are compatible. (The data type with highest precision/length will be chosen.)

  Parameter markers are usually referenced in order by appearance, from left to right. However, if numbers are specified as parameter marker names, these numbers will decide the parameter order.

Examples

```
UPDATE persons
SET last_name = :plastname
WHERE id = :pid;

DELETE FROM persons WHERE id = ?;

SELECT LastName as Name, Address
FROM staff
WHERE City = :cityname
UNION ALL
SELECT companyName, Address
FROM companies
WHERE City = :cityname;

UPDATE persons
SET last_name = :2
WHERE id = :1;

CREATE TABLE tc (x REAL, y DOUBLE PRECISION);
INSERT INTO  tc VALUES (:cval, :cval); -- :cval data type is DOUBLE PRECISION

CREATE TABLE persons (first_name VARCHAR(10), last_name VARCHAR(15));
INSERT INTO persons VALUES (:name, :name); -- :name data type VARCHAR(15)
```

Host Identifiers

Host identifiers are used in SQL statements to identify objects associated with the host language such as variables, declared areas and program statement labels.

Host identifiers are formed in accordance with the rules for forming variable names in the particular host language, see the *Mimer SQL Programmer's Manual, Appendix A, Host Language Dependent Aspects*.

Host identifiers are never enclosed in delimiters and may coincide with SQL reserved words.

The length of host identifiers used in SQL statements may not exceed 128 characters, even if the host language accepts longer names.

Whenever the term `host-variable` appears in the syntax diagrams, one of the three following constructions must be used:

```
:host-identifier1
```

or

```
:host-identifier1 :host-identifier2
```

or

```
:host-identifier1 INDICATOR :host-identifier2
```

`Host-identifier1` is the name of the main host variable.

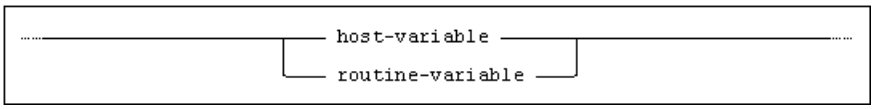
`Host-identifier2` is the name of the indicator variable, used to signal the assignment of a null value to the host variable. See the *Mimer SQL Programmer's Manual, Chapter 4, Indicator Variables,* for a description of the use of indicator variables.

The colon preceding the host identifier serves to identify the variable to the SQL compiler and is not part of the variable name in the host language.

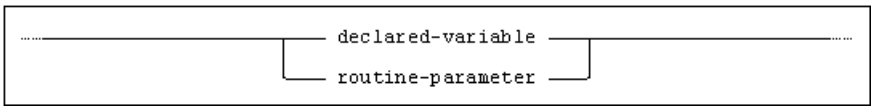
Target Variables

A target variable is an item that may be specified as the object receiving the result of an assignment or a `SELECT INTO`. The objects that may be specified where a target variable is expected differ depending on whether the context is Procedural usage or Embedded usage. For more information, see *Usage Modes* on page *195*.

In the syntax diagrams, replace the term `target-variable`, with the following construction:



where `routine-variable` is:



For more information, see: *DECLARE VARIABLE* on page *315*, *CREATE FUNCTION* on page *258* and *CREATE PROCEDURE* on page *271*.

Note: A `routine-variable` may only be specified in a procedural usage context.

Reserved Words

Appendix A Reserved Words gives a list of keywords reserved in SQL statements. These words must be enclosed in double quotation marks, `""`, if they are used as SQL identifiers.

Example

```
SELECT "MODULE" FROM ...
```

Standard Compliance

This section summarizes standard compliance concerning identifiers.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F391, “Long identifiers”. Feature F392, “Unicode escapes in identifiers”.
	Mimer SQL extension	The use of the special characters $ and # in identifiers is a Mimer SQL extension. Parameter marker as a colon followed by an integer literal is a Mimer SQL extension.

Data Types in SQL Statements

Mimer SQL supports the following data type categories:

- Character strings, see *Character Strings* on page *44*
- National character strings, see *National Character Strings* on page *46*
- Binary, see *Binary* on page *50*
- Numerical, see *Numerical* on page *52*
- Datetime, see *Datetime* on page *53*
- Interval, see *Interval* on page *54*
- Boolean, see *Boolean* on page *57*
- Spatial, see *Spatial Data Types* on page *58*.
- Universally unique identifier (UUID), see *Universally Unique Identifier (UUID)* on page *58*

In SQL statements, you make explicit data type references when creating tables and domains and altering tables. You also use data types in `CAST` and stored procedure variable declarations.

In addition, there is also a `ROW` type that can be used in stored procedures only, for more information see *ROW Data Type* on page *59*.

Character Strings

The character string data types store sequences of bytes that represent alphanumeric data, according to ISO 8859-1.

The character string data type category contains the following data types:

Data Type	Abbreviations	Description	Range
`CHARACTER(n)`	`CHAR(n)`	Character string, fixed length n.	1 <= n <= 15 000
`CHARACTER VARYING(n)`	`CHAR VARYING(n)` `VARCHAR(n)`	Variable length character string, maximum length n.	1 <= n <= 15 000
`CHARACTER LARGE OBJECT (n[K|M|G])`	`CHAR LARGE OBJECT (n[K|M|G])` `CLOB (n[K|M|G])`	Variable length character string measured in characters.	For information on the object length, see *Specifying the CLOB Length* on page *45*.

CHARACTER or CHAR

The CHARACTER (CHAR) data type stores string values of fixed length in a column.

You specify the length of the CHAR data type as the length of the column when you create a table. You can specify the length to be any value between 1 and 15 000.

When Mimer SQL stores values in a column defined as CHAR, it right-pads the values with spaces to conform with the specified column length.

Note: If you define a data type as CHARACTER or CHAR, that is, without specifying a length, the length of the data type is 1.

CHARACTER VARYING or CHAR VARYING or VARCHAR

The CHARACTER VARYING, abbreviated CHAR VARYING or VARCHAR, data type stores strings of varying length.

You specify the maximum length of the VARCHAR data type as the length of the column when you create a table. You can specify the length to be between 1 and 15 000.

CHARACTER LARGE OBJECT or CLOB

The CHARACTER LARGE OBJECT (CLOB) data type stores character string values of varying length up to the maximum specified as the large object length (n[K|M|G]).

The large object length is n, optionally multiplied by K|M|G.

You can specify the maximum length of the CLOB data type as the length of the column when you create the table.

Specifying the CLOB Length

If you specify <n>K (kilo), the length (in characters) is <n> multiplied by 1 024.

If you specify <n>M (mega), the length is <n> multiplied by 1 048 576.

If you specify <n>G (giga), the length is <n> multiplied by 1 073 741 824.

If you do not specify large object length, Mimer SQL assumes that the length of the data type is 1M.

Maximum CLOB Length

The maximum length of a CLOB is determined by the amount of disk space available for its storage.

Using CLOBs

You can work with CLOBs as follows:

- Retrieving CLOBs with simple column references in the SELECT clause of a SELECT statement
- Assigning CLOBs using INSERT statements with a VALUES clause
- Assigning CLOBs using UPDATE statements
- Adding CLOB columns using CREATE TABLE or ALTER TABLE
- Dropping CLOB columns using ALTER TABLE
- Altering CLOB column data types using ALTER TABLE

There are some restrictions associated with using CLOBs. The only comparisons supported for CLOB values are using the NULL predicate and using the LIKE predicate.

The only scalar functions which can be used on CLOB columns are SUBSTRING, CHAR_LENGTH and OCTET_LENGTH.

A CLOB column may not be part of any primary key constraint, index, or unique constraint.

The comparison restrictions also prevent CLOB columns from being used in DISTINCT, GROUP BY and ORDER BY clauses, and UNION, INTERSECT and EXCEPT operations.

When defining a stored procedure or trigger it is not allowed to use a CLOB type for a parameter or a variable. It is allowed to create triggers for tables with CLOB columns with one exception, in an instead of trigger it is not possible to reference CLOB columns in the new table.

Collations

All character strings have a collation attribute. A collation determines the order for ordering and comparisons, see the *Mimer SQL User's Manual, Chapter 4, Collations,* for a detailed description of collations.

National Character Strings

Mimer SQL implements Unicode using the data type NCHAR (i.e. NATIONAL CHARACTER data type). The NCHAR data type is logically UTF-32, however, it is stored in a compressed form. Application host variables may use any of the three encoding forms UTF-8, UTF-16, or UTF-32 when storing NCHAR data in the database. The encoding forms are fully transparent; you may e.g. use UTF-16 to store data, and you can use UTF-8 for fetching data.

The CHAR data type is based on ISO 8859-1 (Latin1), which is a true subset of Unicode, and therefore CHAR and NCHAR are fully compatible.

Normalization

A Unicode character can have several equivalent representations. There are precomposed characters and there are combining characters that can be used together with base characters to form a specific character. Consider the letter E with circumflex and dot below, a letter that occurs in Vietnamese. This letter has five possible representations in Unicode:

- U+0045 LATIN CAPITAL LETTER E
  U+0302 COMBINING CIRCUMFLEX ACCENT
  U+0323 COMBINING DOT BELOW
- U+0045 LATIN CAPITAL LETTER E
  U+0323 COMBINING DOT BELOW
  U+0302 COMBINING CIRCUMFLEX ACCENT
- U+00CA LATIN CAPITAL LETTER E WITH CIRCUMFLEX
  U+0323 COMBINING DOT BELOW
- U+1EB8 LATIN CAPITAL LETTER E WITH DOT BELOW
  U+0302 COMBINING CIRCUMFLEX ACCENT
- U+1EC6 LATIN CAPITAL LETTER E WITH CIRCUMFLEX AND DOT BELOW

Any two of these sequences should compare equal. The Normalization Form C (NFC) of all five sequences is U+1EC6.

In Mimer SQL, Unicode data (NCHAR) is automatically transformed to NFC. When needed, literals and variables are implicitly normalized. The result of a concatenation will always be normalized, and string functions, like UPPER and LOWER, will always return a normalized result string. This will assert that all Unicode data will be in NFC, thus giving the expected result in search operations.

Example

```
SQL>create table t(c nchar(1));
SQL>insert into  t values(u&'E\0302\0323');
SQL>insert into  t values(u&'E\0323\0302');
SQL>insert into  t values(u&'\00CA\0323');
SQL>insert into  t values(u&'\1EB8\0302');
SQL>insert into  t values(u&'\1EC6');
SQL>select count(c) as equal from t where c = u&'\1EC6';

EQUAL
=====
    5
```

The normalization forms are fully described in the Unicode standard annex #15, Unicode Normalization Forms (https://www.unicode.org/reports/tr15).

Case Folding

When converting between upper and lower case most Unicode characters follow a one-to-one case mapping. However, a few characters expand to two or three characters in folding operations.

Folding operations do not always preserve normalization form. In a few instances, the casing operators must normalize after performing their core function.

Consider the following NFC string:

```
U+01F0 LATIN SMALL LETTER J WITH CARON,
U+0323 COMBINING DOT BELOW
```

Its upper case form is:

```
U+004A LATIN CAPITAL LETTER J,
U+030C COMBINING CARON,
U+0323 COMBINING DOT BELOW
```

However, the upper case normalized form (NFC) is:

```
U+004A LATIN CAPITAL LETTER J,
U+0323 COMBINING DOT BELOW,
U+030C COMBINING CARON
```

The Unicode definitions for one-to-one mappings are found here https://www.unicode.org/Public/6.1.0/ucd/UnicodeData.txt, and the expanding definitions are found here https://www.unicode.org/Public/6.1.0/ucd/SpecialCasing.txt.

National Character Data Types

The national character string data type category contains the following data types:

Data Type	Abbreviations	Description	Range
NATIONAL CHARACTER(n)	NATIONAL CHAR(n) NCHAR(n)	National character string, fixed length n.	1 <= n <= 5 000
NATIONAL CHARACTER VARYING(n)	NATIONAL CHAR VARYING(n) NCHAR VARYING(n) NVARCHAR(n)	Variable length, national character string, maximum length n.	1 <= n <= 5 000
NATIONAL CHARACTER LARGE OBJECT(n[K|M|G])	NATIONAL CHAR LARGE OBJECT(n[K|M|G]) NCHAR LARGE OBJECT(n[K|M|G]) NCLOB(n[K|M|G])	Variable length national character string measured in characters.	For information on the object length, see *Specifying the NCLOB Length* on page *49*.

NATIONAL CHARACTER or NATIONAL CHAR or NCHAR

The NATIONAL CHARACTER (NCHAR) data type stores string values of fixed length in a column. You specify the length of the NATIONAL CHARACTER data type as the length of the column when you create a table. You can specify the length to be any value between 1 and 5 000.

When Mimer SQL stores values in a column defined as NATIONAL CHARACTER, it right-pads the values with spaces to conform with the specified column length.

Note: If you define a data type as NATIONAL CHARACTER or NCHAR, that is, without specifying a length, the length of the data type is 1.

NATIONAL CHARACTER VARYING or NATIONAL CHAR VARYING or NCHAR VARYING or NVARCHAR

The NATIONAL CHARACTER VARYING, abbreviated NVARCHAR, NATIONAL CHAR VARYING or NCHAR VARYING, data type stores strings of varying length.

You specify the maximum length of the NATIONAL CHARACTER VARYING data type as the length of the column when you create a table. You can specify the length to be between 1 and 5 000.

NATIONAL CHARACTER LARGE OBJECT or NCLOB

The NATIONAL CHARACTER LARGE OBJECT (NCLOB) data type stores national character string values of varying length up to the maximum specified as the large object length (n[K|M|G]).

The large object length is n, optionally multiplied by K|M|G.

You can specify the maximum length of the NCLOB data type as the length of the column when you create the table.

Specifying the NCLOB Length

If you specify <n>K (kilo), the length (in characters) is <n> multiplied by 1 024.

If you specify <n>M (mega), the length is <n> multiplied by 1 048 576.

If you specify <n>G (giga), the length is <n> multiplied by 1 073 741 824.

If you do not specify large object length, Mimer SQL assumes that the length of the data type is 1M.

Maximum NCLOB Length

The maximum length of an NCLOB is determined by the amount of disk space available for its storage.

Using NCLOBs

You can work with NCLOBs as follows:

- Retrieving NCLOBs with simple column references in the SELECT clause of a SELECT statement
- Assigning NCLOBs using INSERT statements with a VALUES clause
- Assigning NCLOBs using UPDATE statements
- Adding NCLOB columns using CREATE TABLE or ALTER TABLE
- Dropping NCLOB columns using ALTER TABLE
- Altering NCLOB column data types using ALTER TABLE

There are some restrictions associated with using NCLOBs. The only comparison supported for NCLOB values are using the NULL predicate and using the LIKE predicate.

The only scalar functions which can be used on NCLOB columns are SUBSTRING, CHAR_LENGTH and OCTET_LENGTH.

An NCLOB column may not be part of any primary key constraint, index, or unique constraint.

The comparison restrictions also prevent NCLOB columns from being used in DISTINCT, GROUP BY and ORDER BY clauses, and UNION, EXCEPT and INTERSECT operations.

When defining a stored procedure or trigger it is not allowed to use a NCLOB type for a parameter or a variable. It is allowed to create triggers for tables with NCLOB columns with one exception, in an instead of trigger it is not possible to reference NCLOB columns in the new table.

Collations

All national character strings have a collation attribute. A collation determines the order for ordering and comparisons, see *Mimer SQL User's Manual, Chapter 4, Collations* for a detailed description of collations.

Binary

The binary data type stores a sequence of bytes.

The binary data type category contains the following data types:

Data Type	Abbreviation	Description	Range
BINARY(n)	N/A	Fixed length binary string, maximum length n.	1 <= n <= 15 000
BINARY VARYING(n)	VARBINARY(n)	Variable length binary string, maximum length n.	1 <= n <= 15 000
BINARY LARGE OBJECT(n[K|M|G])	BLOB(n[K|M|G])	Variable length binary string measured in octets.	For information on the object length, see *Specifying the BLOB Length* on page *50*.

Note: How binary data is displayed depends on the SQL tool used. For example, Mimer BSQL displays binary data as its hexadecimal value.

BINARY LARGE OBJECT or BLOB

The BINARY LARGE OBJECT or BLOB data type stores binary string values of varying length up to the maximum specified as the large object length (n[K|M|G]).

The large object length is n, optionally multiplied by K|M|G.

Data stored in BLOB's may only be stored in the database and retrieved again, it cannot be used in arithmetical operations.

Specifying the BLOB Length

If you specify <n>K, the length is <n> multiplied by 1 024.

If you specify <n>M, the length is <n> multiplied by 1 048 576.

If you specify <n>G, the length is <n> multiplied by 1 073 741 824.

If you do not specify large object length, Mimer SQL assumes that the length of the data type is 1M.

Maximum BLOB Length

The maximum length of a BLOB is determined by the amount of disk space available for its storage.

Using BLOBs

You can work with BLOB's as follows:

- Retrieving BLOB's with simple column references in the SELECT clause of a SELECT statement
- Assigning BLOB's using INSERT statements with a VALUES clause
- Assigning BLOB's using UPDATE statements

- Adding BLOB columns using CREATE TABLE or ALTER TABLE
- Dropping BLOB columns using ALTER TABLE
- Altering BLOB column data types using ALTER TABLE

There are some restrictions associated with using BLOB's. The only comparison supported for BLOB values is using the NULL predicate and using the LIKE predicate.

The only scalar functions which can be used on BLOB columns are SUBSTRING, CHAR_LENGTH and OCTET_LENGTH.

A BLOB column may not be part of any primary key constraint, index, or unique constraint.

The comparison restrictions also prevent BLOB columns from being used in DISTINCT, GROUP BY and ORDER BY clauses and UNION, EXCEPT and INTERSECT statements.

When defining a stored procedure or trigger it is not allowed to use a BLOB type for a parameter or a variable. It is allowed to create triggers for tables with BLOB columns with one exception, in an instead of trigger it is not possible to reference BLOB columns in the new table.

Numerical

The numerical data type category contains the following data types:

Data type	Abbreviation	Description	Range
`SMALLINT`	N/A	Integer numerical, precision 5.	-32 768 through 32 767 Corresponds to a 2 bytes, signed int.
`INTEGER`	`INT`	Integer numerical, precision 10.	-2 147 483 648 through 2 147 483 647 Corresponds to a 4 bytes, signed int.
`BIGINT`	N/A	Integer numerical, precision 19.	-9 223 372 036 854 775 808 through 9 223 372 036 854 775 807 Corresponds to an 8 bytes, signed int.
`INTEGER(p)`	`INT(p)`	Integer numerical, precision p.	1 <= p <= 45
`DECIMAL(p,s)`	`DEC(p,s)`	Exact numerical, precision p, scale s.	1 <= p <= 45 0 <= s <= p
`REAL`	N/A	Floating point value with 24-bit binary mantissa.	Zero or absolute value from 1.40129846^{-45} to 3.40282347^{+38} Corresponds to single precision float.
`DOUBLE PRECISION`	N/A	Floating point value with 53-bit binary mantissa.	Zero or absolute value from $4.9406564584124654^{-324}$ to $1.7976931348623157^{+308}$ Corresponds to double precision float.
`FLOAT`	N/A	Floating point value with 53-bit binary mantissa.	Zero or absolute value from $4.9406564584124654^{-324}$ to $1.7976931348623157^{+308}$ Corresponds to double precision float. Same as DOUBLE PRECISION.
`FLOAT(p)`	N/A	Floating point value with p digits in the decimal mantissa.	1 <= p <= 45 Zero or absolute value 10^{-999} to 10^{+999}

All numerical data may be signed.

For all numerical data, the precision p indicates the maximum number of decimal digits the number may contain, excluding any sign or decimal point.

For decimal data, the scale s indicates the fixed number of digits following the decimal point.

Note: The decimal data with scale zero `DECIMAL(p,0)` is not the same as integer `INTEGER(p)`.

For `FLOAT(p)`, floating point (approximate numerical) data is stored in exponential form. The precision is specified for the mantissa only. The permissible range of the exponent is -999 to +999.

Specifying Data Type Precision and Scale

In the following cases, the omission of scale, or the omission of both precision and scale, is allowed (scale may not be specified without precision):

Data Type	Abbreviation	
`DECIMAL`	`DEC`	is equivalent to `DECIMAL(15,0)`
`DECIMAL(5)`	`DEC(5)`	is equivalent to `DECIMAL(5,0)`

Note: The data type `INTEGER` is distinct from `INTEGER(10)`. (`INTEGER(10)` may store values between -9 999 999 999 and 9 999 999 999, but `INTEGER` may only store values between -2 147 483 648 and 2 147 483 647.)

Datetime

`DATETIME` is a term used to collectively refer to the data types `DATE`, `TIME(s)` and `TIMESTAMP(s)`.

Data type	Description

`DATE`
`TIME(s)`
`TIMESTAMP(s)`
	Composed of a number of integer fields, represents an absolute point in time, depending on sub-type.
	Default `s` value is 0 for `TIME` and 6 for `TIMESTAMP`.

DATE

`DATE` describes a date using the fields `YEAR`, `MONTH` and `DAY` in the format `YYYY-MM-DD`. It represents an absolute position on the timeline.

TIME(s)

`TIME(s)` describes a time in an unspecified day, with seconds precision *s*, using the fields `HOUR`, `MINUTE` and `SECOND` in the format `HH:MM:SS[.sF]` where `F` is the fractional part of the `SECOND` value. It represents an absolute time of day.

TIMESTAMP(s)

`TIMESTAMP(s)` describes both a date and time, with seconds precision `s`, using the fields `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE` and `SECOND` in the format `YYYY-MM-DD HH:MM:SS[.sF]` where `F` is the fractional part of the `SECOND` value. It represents an absolute position on the timeline.

DATETIME Significance

A DATETIME contains some or all of the fields YEAR, MONTH, DAY, HOUR, MINUTE and SECOND. These fields always occur in the order listed, which is from the most significant to least significant. Year is the most significant.

Each of the fields is an integer value, except that the SECOND field may have an additional integer component to represent the fractional seconds.

For a DATETIME value with a SECOND component, it is possible to specify an optional seconds precision which is the number of significant digits in the fractional part of the SECOND value. This must be a value between 0 and 9. If a SECOND's precision is not specified, the default is 0 for TIME and 6 for TIMESTAMP.

Calendar and Clock

DATE values are represented according to the Gregorian calendar. TIME values are represented according to the 24 hour clock.

Inclusive Value Limits for DATETIME

The inclusive value limits for the DATETIME fields are as follows:

Field	Inclusive value limit
YEAR	0001 to 9999
MONTH	01 to 12
DAY	01 to 31 (upper limit further constrained by MONTH and YEAR)
HOUR	00 to 23
MINUTE	00 to 59
SECOND	00 to 59.999999999

Interval

An INTERVAL is a period of time, such as: 3 years, 90 days or 5 minutes and 45 seconds.

Data Type	Description
INTERVAL	Composed of a number of integer fields, represents a period of time, depending on the type of interval.

There are effectively two kinds of INTERVAL:

- **YEAR-MONTH**
  containing one or both of the fields YEAR and MONTH. (Also known as long interval.)
- **DAY-TIME**
  containing one or more consecutive fields from the set DAY, HOUR, MINUTE and SECOND. (Also known as short interval.)

The distinction is made between the two interval types in order to avoid the ambiguity that would arise if a MONTH value was combined with a field of lower significance, e.g. DAY, given that different months contain differing numbers of days.

For example, the hypothetical interval 2 months and 10 days could vary between 69 and 72 days in length, depending on the months involved. Therefore, to avoid unwanted variations in the downstream arithmetic etc. the variable length MONTH component may only exist at the lowest significance level in an INTERVAL.

The SECOND field may also only exist at the lowest significance level in an INTERVAL, simply because it is the least significant of all the fields.

An INTERVAL data type is a signed numeric quantity (i.e. negative INTERVALs are allowed) comprising a specific set of fields. The list of fields in an INTERVAL is called the interval precision.

The fields in an INTERVAL are exactly the same as those previously described for DATETIME except that the value constraints imposed on the most significant field are determined by the leading precision (*p* in *Interval Qualifiers* on page *55*) for the INTERVAL type and not by the Gregorian calendar and 24 hour clock.

A leading precision value between 1 and the maximum allowed for the field type may be specified for an INTERVAL. If none is specified, the default is 2.

Value Constraints for Fields in an Interval

The table below shows the maximum permitted leading precision values for each field type in an INTERVAL:

Field	Maximum leading precision
YEAR	7
MONTH	7
DAY	7
HOUR	8
MINUTE	10
SECOND	12

The value of a MONTH field, which is not in the leading field position, is constrained between 0 and 11, inclusive, in an INTERVAL (and not between 1 and 12 as in a DATETIME).

Where the SECOND field is involved, seconds precision (*s* in *Interval Qualifiers* on page *55*) can be specified for it in the same way as for DATETIME.

Note that in the INTERVAL consisting only of a SECOND field (INTERVAL SECOND), the SECOND field will have both a leading precision and a seconds precision, specified together.

A seconds precision value between 0 and 9 may be specified for an INTERVAL. If the seconds precision is not specified, a default value of 6 is implied.

Interval Qualifiers

A syntactic element, the interval qualifier, is used to specify the interval precision, leading precision and (where appropriate) the seconds precision.

The interval qualifier follows the keyword INTERVAL when specifying an INTERVAL data type.

The following table lists the valid interval qualifiers for YEAR-MONTH intervals:

Interval Qualifier	Range	Description
YEAR*(p)*	1 <= p <= 7	An interval class describing a number of years, with a leading precision *p*. It contains a YEAR field in the format: *p*Y. Default precision is 2.
MONTH*(p)*	1 <= p <= 7	An interval class describing a number of months, with leading precision *p*. It contains a MONTH field in the format: *p*M. Default precision is 2.
YEAR*(p)* TO MONTH	1 <= p <= 7	An interval class describing a number of years and months, with leading precision *p*. The format is: *p*Y-MM. Default precision is 2.

The following table lists the valid interval qualifiers for DAY-TIME intervals:

Interval Qualifier	Range	Description
DAY*(p)*	1 <= p <= 7	An interval class describing a number of days, with a leading precision *p*. It contains a DAY field in the format: *p*D. Default precision is 2.
HOUR*(p)*	1 <= p <= 8	An interval class describing a number of hours, with leading precision *p*. It contains an HOUR field in the format: *p*H. Default precision is 2.
MINUTE*(p)*	1 <= p <= 10	An interval class describing a number of minutes, with leading precision *p*. It contains a MINUTE field in the format: *p*M. Default precision is 2.
SECOND*(p,s)*, SECOND*(p)*	1 <= p <= 12, 0 <= s <= 9	An interval class describing a number of seconds, with leading precision *p* and seconds precision *s*. It contains a SECOND field in the format: *p*S[.*s*F]. (F is the fractional part of the seconds value.) Default precision is 2, default scale is 6.
DAY*(p)* TO HOUR	1 <= p <= 7	An interval class describing a number of days and hours, with leading precision *p*. The format is: *p*D HH. Default precision is 2.

Interval Qualifier	Range	Description
DAY*(p)* TO MINUTE	1 <= p <= 7	An interval class describing a number of days, hours and minutes, with leading precision *p*. The format is: *p*D HH:MM. Default precision is 2.
DAY*(p)* TO SECOND*(s)*	1 <= p <= 7	An interval class describing a number of days, hours, minutes and seconds, with leading precision *p*. The format is: *p*D HH:MM:SS[.*s*F]. Default precision is 2, default scale is 6.
HOUR*(p)* TO MINUTE	1 <= p <= 8	An interval class describing a number of hours and minutes, with leading precision *p*. The format is: *p*H:MM. Default precision is 2.
HOUR*(p)* TO SECOND*(s)*	1 <= p <= 8, 0 <= s <= 9	An interval class describing a number of hours, minutes and seconds, with leading precision *p* and seconds precision *s*. The format is: *p*H:MM:SS[.*s*F]. Default precision is 2, default scale is 6.
MINUTE*(p)* TO SECOND*(s)*	1 <= p <= 10, 0 <= s <= 9	An interval class describing a number of minutes and seconds, with leading precision *p* and seconds precision *s*. The format is: *p*M:SS[.*s*F]. Default precision is 2, default scale is 6.

Length of an Interval Data Type

The length of an INTERVAL data type is the same as the number of characters required to represent it as a string and is determined by the interval precision, leading precision and the seconds precision (where it applies).

The maximum length of an INTERVAL data type can be computed according to the following rules:

- The length of the most significant field is the leading precision value (*p*).
- Allow a length of 2 for each field following the most significant field.
- Allow a length of 1 for each separator between fields. Separators occur between YEAR and MONTH, DAY and HOUR, HOUR and MINUTE, and MINUTE and SECOND.
- If seconds precision applies, and is non-zero, allow a length equal to the seconds precision value, plus 1 for the decimal point preceding the fractional part of the seconds value.

Boolean

BOOLEAN describes a truth value. It can have the values TRUE or FALSE.

Spatial Data Types

The spatial data types can be used for geographical data (longitude, latitude and location), and for coordinate system data (x, y, coordinate).

The following user-defined types are used to store spatial data:

Type	SQL type	Description
BUILTIN.GIS_LATITUDE	BINARY(4)	A distinct user-defined type that stores latitude values. See *Mimer SQL Programmer's Manual, Appendix 14, BUILTIN.GIS_LATITUDE*.
BUILTIN.GIS_LONGITUDE	BINARY(4)	A distinct user-defined type that stores longitude values. See *Mimer SQL Programmer's Manual, Appendix 14, BUILTIN.GIS_LONGITUDE*.
BUILTIN.GIS_LOCATION	BINARY(8)	A distinct user-defined type that is used to store a location on Earth. It has a latitude and a longitude component. See *Mimer SQL Programmer's Manual, Appendix 14, BUILTIN.GIS_LOCATION*.
BUILTIN.GIS_COORDINATE	BINARY(8)	This type has an x and a y component in a flat coordinate system. See *Mimer SQL Programmer's Manual, Appendix 14, BUILTIN.GIS_COORDINATE*.

See *Mimer SQL Programmer's Manual, Appendix 14, Spatial Data* for a description of the GIS (Geographic information system) functionality.

Universally Unique Identifier (UUID)

The following type can be used for storing universally unique identifier (UUID) data:

Type	SQL type	Description
BUILTIN.UUID	BINARY(16)	A distinct user-defined type for storing uuid values.

See *Mimer SQL Programmer's Manual, Appendix 15, Universally Unique Identifier - UUID* for more information.

ROW Data Type

There is an additional data type supported by Mimer SQL, called the `ROW` data type, which is used in stored procedures only.

A variable which is declared as having the `ROW` data type implicitly defines a row value, which is a single construct that has a value which effectively represents a table row.

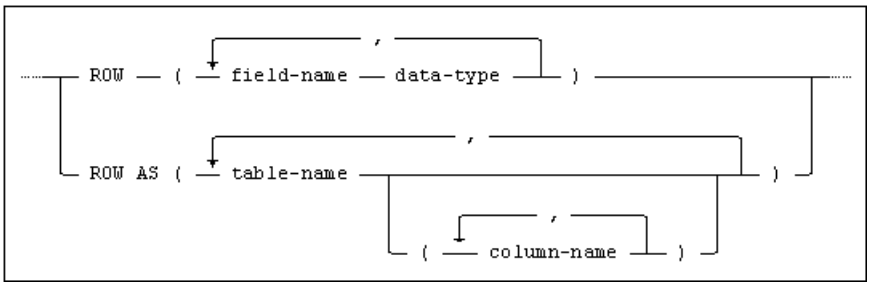
A row value is composed of a number of named values, each of which has its own data type and represents a column value in the overall row value.

A `ROW` data type can be defined either by explicitly specifying a number of field-name/data-type pairs or by specifying a number of table columns from which the unqualified names and data types are inherited.

A `ROW` data type definition can be specified where one of the above data types would normally be used in a variable declaration in a compound statement, see the *Mimer SQL Programmer's Manual, Chapter 11, The ROW Data Type,* for details.

ROW Data Type Syntax

The syntax for defining a ROW data type is:



The following points apply to the specification of a `ROW` data type:

- The value specified for `data-type` can be a `ROW` data type specification.
- Two fields in the same `ROW` data type specification must not have the same name (this restriction applies equally to fields named by specifying a `field-name` value and those named by inheriting the unqualified name of a table column).
- If `table-name` is specified without a list of column names, all the columns in the table are used to define fields in the `ROW` data type.

Note: If a row specification uses the `AS` clause, any check constraints for the table will not be validated. If a field is declared as using a domain, the same behavior as for a single variable will occur.

The Null Value

Columns which contain an undefined value are assigned a null value.

Depending on the context, this is represented in SQL statements either by the keyword `NULL` or by a host variable associated with an indicator variable whose value is minus one, see the *Mimer SQL Programmer's Manual, Chapter 4, Indicator Variables*.

The null value is generally never equal to any value, not even to itself. All comparisons involving null evaluate to unknown, see *Comparisons* on page *80*.

Note: Null values are treated as equal to each other for the purposes of `DISTINCT`, `GROUP BY`, `ORDER BY`, `UNION`, `INTERSECT`, `EXCEPT` and `IS [NOT] DISTINCT FROM`.

Null values are sorted at the end of ascending sequences and at the beginning of descending sequences.

Data Type Compatibility

Assignment and comparison operations generally require that the data types of the items involved (literals, variables or column values) are compatible but not necessarily exactly equivalent.

Any exceptions to this rule are specified in the detailed syntax descriptions in *Chapter 12, SQL Statements*.

All character data is compatible with all other character data.

Numerical data is compatible with other numerical data regardless of specific data type (integer, decimal or float). Rules for operations involving mixed numerical data types are described in *Comparisons* on page *80*.

Datetime and interval data types can be combined in arithmetic operations, for details, see *Datetime and Interval Arithmetic* on page *60*.

Values stored in host variables (but not literals or column values) may be converted between character and numerical data types if required by the operation using the variable. The declared type of the variable itself is not altered.

Similarly, character columns may be assigned to numerical variables and vice versa. The rules for data type conversion are given below.

Variables may be converted between different data types by using the `CAST` function.

Datetime and Interval Arithmetic

The following table lists the arithmetic operations that are permitted involving `DATE`, `TIME`, `TIMESTAMP` (`DATETIME`) or `INTERVAL` values:

Operand 1	Operator	Operand 2	Result Type
DATETIME	-	DATETIME	(See discussion below)
DATETIME	+ or -	INTERVAL	DATETIME
INTERVAL	+	DATETIME	DATETIME
INTERVAL	+ or -	INTERVAL	INTERVAL
INTERVAL	* or /	NUMERIC	INTERVAL
NUMERIC	*	INTERVAL	INTERVAL

Operands can not be combined arithmetically unless their data types are comparable, see *Comparisons* on page *80*. If either operand is the null value, then the result will always be the null value.

If an arithmetic operation involves two `DATETIME` or `INTERVAL` values with a defined scale, the scale of the result will be the larger of the scales of the two operands.

When an `INTERVAL` value is multiplied by a numeric value, the scale of the result is equal to that of the `INTERVAL` and the precision of the result is the leading precision of the `INTERVAL` increased by 1. In the case of division, the same is true except that the precision of the result is equal to the leading precision of the `INTERVAL` (i.e. it is not increased by 1).

When two INTERVAL values are added or subtracted, the scale (*s*) and precision (*p*) of the result are described by the following rule:

```
p = min(MLP, max(p'-s', p"-s") + max(s', s") + 1)
s = max(s', s")
```

where MLP is the maximum permitted leading precision for the INTERVAL type of the result, refer to the table in *Interval* on page *54* for these values.

The interval precision of the result is the combined interval precision of the two operands, e.g.

```
DAY TO HOUR + MINUTE TO SECOND
```

will produce a DAY TO SECOND result.

One DATETIME value may be subtracted from another to produce an INTERVAL that is the signed difference between the stated dates or times.

The application must, however, specify an INTERVAL date type for the result by using an interval-qualifier.

Thus, the syntax is:

```
(DATETIME1 - DATETIME2) interval-qualifier
```

Example:

```
(DATE '2016-01-09' - DATE '2016-01-01') DAY
```

This, therefore, evaluates to INTERVAL '8' DAY.

Host Variable Data Type Conversion

When a host variable is used in assignments, comparisons or expressions where the data type of the variable is different from the data type of literals or column declarations, an attempt is made internally to convert the value of the variable to the appropriate type.

Character and Character

Conversion between a character variable and a character value is always allowed. The conversion follows these rules:

- When assigning a character value to a character variable, where the variable is longer than the character value, the variable is padded with trailing blanks.
- When assigning a character value to a character variable, where the value is longer than the variable, the value is truncated and a warning status is returned. If only blanks are truncated, no warning is returned.
- When assigning a variable length character, i.e. a VARCHAR or NCHAR VARYING, column from a character variable, the column is padded with blanks up to the length of the character variable if the column is longer than the variable.
- When assigning a variable length character column from a character variable, where the column is shorter than the variable (except for trailing spaces), the assignment will fail and an error message is returned.

National Character and Character

- When assigning a national character column to a character variable, characters outside the Latin1 character set may occur.
- When assigning a character column to a wide character variable, all characters will be converted to the wide character format.
- When assigning a character column a national character value where characters outside the Latin1 character set occur, the assignment will fail and an error message is returned.
- When assigning a character value to a national character column, the value will be converted to the national character data type.

Numerical and Character

Numerical values may always be converted to character strings, provided that the character string variable is sufficiently long enough. The resulting string format is illustrated below, using n to represent the appropriate number of digits and s to represent the sign position (a minus sign for negative values).

Two digits are always used for the exponent derived from REAL numbers, and three digits are used for all other floating point numbers, regardless of the value of the exponent. The sign of the exponent is always given explicitly (+ or -).

Numerical data	String length	String format
Integer numerical precision p	p+1	'sn'
Exact numerical precision p, scale s	p+2	'sn.n'
REAL	15	'sn.nnnnnnnnEsnn'
DOUBLE PRECISION	24	'sn.nnnnnnnnnnnnnnnnEsnnn'
FLOAT	24	'sn.nnnnnnnnnnnnnnnnEsnnn'
FLOAT(p)	p+7	'sn.nEsnnn'

Note: Decimal values with scale 0 are converted to strings with the format 'sn.'. Decimal values where the scale is equal to the precision result in strings with the format 's.n'.

Examples of Assignment Results

Value	Type	Character value
1342	`INTEGER`	'1342'
-15	`INTEGER`	'-15'
13.42	`DECIMAL(6,4)`	'13.4200'
-13.	`DECIMAL(5,0)`	'-13.'
.13	`DECIMAL(2,2)`	'.13'
-1.3E56	`DOUBLE PRECISION`	'-1.30000000000000E+056'

Only numerical character strings can be converted to numerical data.

Numerical strings are defined as follows:

- **Integer**

  One optional sign character (+ or -) followed by at least one digit (0-9). Leading and trailing blanks are ignored. No other character is allowed.

- **Decimal**

  As integer, but with one decimal point (.) placed immediately before or after a digit.

- **Float**

  As decimal, but followed directly by an uppercase or lowercase letter E and an exponent written as an integer (optionally signed).

The precision and scale of a number derived from a numerical character string follows the format of the string.

Leading and trailing zeros are significant for assigning precision.

Thus:

Numerical value	Type
3	`INTEGER(1)`
003	`INTEGER(3)`
0.3	`DECIMAL(2,1)`
00.30	`DECIMAL(4,2)`
.3	`DECIMAL(1,1)`
-33	`INTEGER(2)`
-33.	`DECIMAL(2,0)`
003.3E14	`FLOAT(4)`

Standard Compliance

This section summarizes standard compliance concerning data types.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F052, “Intervals and datetime arithmetic” support for interval data type. Feature F421, “National character” support for national character data type NCHAR and NCHAR VARYING. Feature F555, “Enhanced seconds precision” support for time and timestamps with fraction of seconds. Feature T021, “BINARY and VARBINARY data types”. Feature T031, “Boolean data type” Feature T041, “Basic LOB data type support” Feature T071, “Bigint data type”
	Mimer SQL extension	Conversion between character and numeric when storing values from or retrieving values into host variables is a Mimer SQL extension. Support for the abbreviation NVARCHAR is a Mimer SQL extension. Specifying a precision for INTEGER is a Mimer SQL extension.

Literals

Literal, i.e. fixed data, values may be given for any of the data types supported in SQL statements, wherever the term `literal` appears in the syntax diagrams.

String Literals

A string literal may be represented as a `character-string-literal`, a `national-character-string-literal`, or a `unicode-character-string-literal`.

- **Character-string-literal**

  A `character-string-literal` consists of a sequence of characters enclosed in string delimiters. The standard string delimiter is the single quotation mark: `'`. Two consecutive single quotation marks within a string are interpreted as a single quotation mark.

  If characters outside the ISO 8859-1 character set (Latin1) is included in a `character-string-literal`, the literal will be considered as a `national-character-string-literal`.

  Note: An empty string (i.e. `''`) is a defined value. (It is not a null value.)

- **National-character-string-literal**

  A `national-character-string-literal` consists of a sequence of Unicode characters enclosed in string delimiters and preceded by the optional letter `N`. (I.e. if the `N` letter is missing, and the string literal still contains non Latin1 characters, the literal is a `national-character-string-literal`.)

The standard string delimiter is the single quotation mark: '. Two consecutive single quotation marks within a string are interpreted as a single quotation mark. The case of the preceding N is irrelevant.

Note: An empty string (i.e. N'') is a defined value. (It is not a null value.)

- **Unicode-string-literal**

  A unicode-string-literal is used in order to facilitate the specification of Unicode characters in an ASCII environment. It consists of a sequence of Unicode characters enclosed in string delimiters and preceded by the letter U and an ampersand, i.e. U&. The standard string delimiter is the single quotation mark: '. Two consecutive single quotation marks within a string are interpreted as a single quotation mark. Unicode characters are given by four hexadecimal digits preceded by a backslash character (\) or, by six hexadecimal digits preceded with a backslash character and a plus character. Two consecutive backslash characters within a string are interpreted as a single backslash character. The case of the preceding U is irrelevant.

  Note: An empty string (i.e. U&'') is a defined value. (It is not a null value.)

Character Separators

For character, national-character, unicode and hexadecimal-string-literals, you can use a separator within the literal to join two or more substrings. Separators are described in *Special Characters* on page *37*.

This is particularly useful when a string literal extends over more than one physical line, or when control codes are to be combined with character sequences.

Examples

ASCII codes are used for the hexadecimal literals:

String	Value
'ABCD'	ABCD
'Mimer''s'	Mimer's
'data'<LF>'base'	database
X'0D0A09'	<CR><LF><TAB>
X'0D0A'<LF>'09'	<CR><LF><TAB>
U&'Malm\00F6'	Malmö

Note: Since a hexadecimal-string is of type binary, an explicit CAST is required when using a hexadecimal-string as character data. For CAST information, see *Assignments* on page *77*.

Numerical Integer Literals

A numerical integer literal is a signed or unsigned number that does not include a decimal point. The sign is a plus (+) or minus (-) sign preceding the first digit.

In determining the precision of an integer literal, leading zeros are significant (i.e. the literal 007 has precision 3).

Examples:

```
  47
-125
+006
   0
```

Numerical Decimal Literals

A numerical decimal literal is a signed or unsigned number containing exactly one decimal point.

In determining the precision and scale of a decimal literal, both leading and trailing zeros are significant (i.e. the literal 003.1400 has precision 7, scale 4).

Examples:

```
   4.7
  -3.
+012.067
   0.0
    .370
```

Numerical Floating Point Literals

Floating point literals are represented in exponential notation, with a signed or unsigned integer or decimal mantissa, followed by an letter E, followed in turn by a signed or unsigned integer exponent.

The base for the exponent is always 10. The exponent zero may be used. The case of the letter E is irrelevant.

In determining the precision of a floating point literal, leading zeros in the mantissa are significant (i.e. the literal 007E4 has precision 3).

Examples:

```
1.3E5    means 130000
-4e-2    means     -0.04
+03.3E2  means    330
0E+45    means      0
1.53E00  means      1.53
```

REAL, DOUBLE PRECISION and FLOAT Literals

There is no syntax for specifying a REAL, DOUBLE PRECISION or FLOAT literal directly.

Instead use a numerical literal specifying an integer, decimal or floating point value. This value can be cast explicitly to REAL, DOUBLE PRECISION or FLOAT by using a `CAST` construct. If the literal is used in a position where a REAL, DOUBLE PRECISION or FLOAT value is expected, an implicit `CAST` is used.

Examples:

```
INSERT INTO TAB(REALCOL) VALUES (20000001); -- Implicit cast
SET ? = CAST(0.1 as DOUBLE PRECISION); -- Explicit cast
```

Note that values of type REAL, DOUBLE PRECISION or FLOAT have a binary mantissa. It is not always possible to store the exact decimal value in those types. In such cases the nearest value will be used. In both cases above, the literal value will be silently rounded.

DATE, TIME and TIMESTAMP Literals

A literal that represents a `DATE`, `TIME` or `TIMESTAMP` value consists of the corresponding keyword shown below, followed by text enclosed in single quotes (`''`).

The following formats are allowed:

```
DATE 'date-value'
TIME 'time-value'
TIMESTAMP 'date-value <space> time-value'
```

A `date-value` has the following format:

```
year-value  -  month-value  -  day-value
```

A `time-value` has the following format:

```
hour-value  :  minute-value  :  second-value
```

where `second-value` has the following format:

```
whole-seconds-value [. fractional-seconds-value]
```

The `year-value`, `month-value`, `day-value`, `hour-value`, `minute-value`, `whole-seconds-value` and `fractional-seconds-value` are all unsigned integers.

A `year-value` contains exactly 4 digits, a `fractional-seconds-value` may contain up to 9 digits and all the other components each contain exactly 2 digits.

Examples:

```
DATE '2017-02-19'
TIME '10:59:23'
TIMESTAMP '2018-11-05 19:20:23.4567'
TIMESTAMP '2021-12-31 23:59:30'
```

Interval Literals

An interval literal represents an interval value and consists of the keyword `INTERVAL` followed by text enclosed in single quotes, in the following format:

```
INTERVAL '[+|-]interval-value' interval-qualifier
```

The `interval-value` text must be a valid representation of a value compatible with the `INTERVAL` data type specified by the `interval-qualifier`, see *Interval Qualifiers* on page *55*.

- If the interval precision includes the `YEAR` and `MONTH` fields, the values of these fields should be separated by a minus sign.
- If the interval precision includes the `DAY` and `HOUR` fields, the values of these fields should be separated by a space.

- If the interval precision includes the HOUR fields and another field of lower significance (MINUTE and/or SECOND), the values of these fields should be separated by a colon.
- All fields may contain up to 2 digits except that:
  - The number of digits in the most significant field must not exceed the leading precision explicitly defined by the interval-qualifier. If a leading precision is not explicitly specified in the interval-qualifier, the default (2) applies.
  - The SECOND field may have a fractional part, whose maximum length is defined by the interval-qualifier.

Examples:

```
INTERVAL '1:30' HOUR TO MINUTE
INTERVAL '1-6' YEAR TO MONTH
INTERVAL '1000 10:20:30.123' DAY(4) TO SECOND(3)
INTERVAL '-199' YEAR(3) **evaluates to -199
INTERVAL '199' YEAR(2) **Invalid : leading precision is 2
INTERVAL '5.555' SECOND(1,2) **evaluates to 5.55
INTERVAL '-5.555' SECOND(1,2) **evaluates to -5.55
INTERVAL '19 23' DAY TO MINUTE **Invalid : no minutes in literal
```

Binary Literals

A binary literal represents an binary value, and is specified as a hexadecimal string.

- **Hexadecimal-string-literal**

  A hexadecimal-string-literal is a string specified as a sequence of hexadecimal values, enclosed in single quotation marks and preceded by the letter X. The sequence of values must contain an even number of positions (every character in the string literal is represented by a two-position value), and may not contain any characters other than the digits 0-9 and the letters A-F. The case of letters (and of the preceding X) is irrelevant. The code values for characters are those which apply in the host system.

Examples:

```
X'5A65794B697A'
x'f66c'
```

Boolean literals

A boolean literal represents a truth value. There are two boolean literals, TRUE and FALSE.

Boolean literals can be used when assigning values and making comparisons, e.g.

```
UPDATE methods SET isConstructor = TRUE WHERE methodName = 'PERSON'

DECLARE v_amountPaid,v_amountDue DECIMAL(10,2);
DECLARE v_isPaid BOOLEAN DEFAULT FALSE;
```

```
SET v_isPaid = v_amountPaid >= v_amountDue;
IF v_isPaid IS TRUE THEN
```

In the last example the comparison with `TRUE` is not needed. The statement can be written as:

```
IF v_isPaid THEN
```

Note: Do not enclose boolean literals in string delimiters. `'TRUE'` is a string literal, not a boolean literal.

Spatial literals

The spatial data types are implemented as user-defined types, with functions to create instances, and methods to return the values in different formats. For more information, see *Mimer SQL Programmer's Manual, Appendix 14, Spatial Data*.

Standard Compliance

This section summarizes standard compliance concerning literals.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core SQL	Feature T021, "BINARY and VARBINARY data types". Feature T031, "BOOLEAN data type".
	Mimer SQL extension	The presence of a newline character (<LF>) between substrings in a character- or hexadecimal-string-literal is not mandatory in Mimer SQL.

Chapter 7

Operators and Values

This chapter discusses operators, value specifications and default values in Mimer SQL. It also discusses assignments, comparisons and result data types.

Operators

Operators manipulate individual data items (operands) and return a result. Mimer SQL uses the following operators:

Set Operators

UNION or UNION ALL

Derives a final result set by combining two other result sets.

If you specify `UNION ALL`, the result consists of all rows in both results sets.

If you only specify `UNION`, the final result set is the set of all rows in both of the result sets, with duplicate rows removed.

See *The UNION Operator* on page *184* for more information.

EXCEPT or EXCEPT ALL

The except operator is used to combine two result sets to one where the combined result set is all records from the first result which is not present in the second result set. If except is specified without the `ALL` quantifier, duplicates are removed from the combined result set. If `ALL` is specified, duplicates are not removed.

See *The EXCEPT Operator* on page *185* for more information.

INTERSECT or INTERSECT ALL

The intersect operator is used to combine two result sets to one where the combined result set is the records that are present in both result sets. If intersect is specified without the `ALL` quantifier, duplicates are removed from the combined result set. If `ALL` is specified, duplicates are not removed.

See *The INTERSECT Operator* on page *185* for more information.

Arithmetical Operators

Arithmetical operators are used in forming expressions, see *Expressions* on page *141*.

The operators are:

- unary arithmetical (i.e. one argument operators)
- binary arithmetical (i.e. two argument operators)

Unary Arithmetical

`+` leaves operand unchanged

`-` changes sign of operand

Binary Arithmetical

`+` addition

`-` subtraction

`*` multiplication

`/` division

`%` modulo

String Operators

String operators are used in forming expressions, see *Expressions* on page *141*.

String

`||` concatenation

Bit Operators

Bit operations:

`&` bitwise AND operation

`|` bitwise OR operation

`^` bitwise XOR operation

`~` bitwise complement

<< left shift

>> right shift

Bit operations are supported on integer data types, i.e. the operations are performed on the integer bit representation. When right shift is performed a so called arithmetic shift is performed. This means that the sign bit will be used to replace the bits as the shift to the right is made.

Examples

Expression	Result
`set ? =  9 | 3`	`11`
`set ? = cast(x'0f23' as int) & cast(x'fff0' as int)`	`3872`
`set ? = cast(cast(x'0f23' as int) & cast(x'fff0' as int) as binary(4))`	`X'00000F20'`
`select cast(~(1 << 7) as binary(8)) from information_schema.ext_onerow`	`X'FFFFFFFFFFFFFF7F'`

Comparison Operators

Comparison operators are used tocompare operands in basic and quantified predicates. (Relational operators are used to compare operands in all other predicates, see *Predicates* on page *153*.)

Both comparison and relational operators perform essentially similar functions. However, comparison operators are common to most programming languages, while the relational operators are more or less specific to SQL.

Comparison Operators

Comparison operator	Explanation
=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Quantifiers

Quantifier
ALL
SOME
ANY

Logical Operators

Logical operator
AND
OR
NOT

The operators AND and OR are used to combine several predicates to form search conditions, see *Search Conditions* on page *165*.

The operator NOT may be used to reverse the truth value of a predicate in forming search conditions. This operator is also available in predicate constructions to reverse the function of a relational operator, see *Search Conditions* on page *165*.

Operator Precedence

Category	SQL
Postfix	(,)
Unary, complement	+, -, ~
Multiplicative	*, /, %
Additive, concatenation	+, -, ||
Shift	<<, >>
Bitwise AND	&
Bitwise OR	|
Comparison operators Boolean test Predicates	=, <>, <, <=, >, >= IS [NOT] FALSE/TRUE/UNKNOWN BETWEEN, LIKE, IS [NOT] NULL, DISTINCT FROM, OVERLAPS, ALL, ANY, SOME, IN, EXISTS, UNIQUE
Negation	NOT
Boolean AND	AND
Boolean OR	OR
Assignment	=

Operators with the same precedence are evaluated from left to right.

Examples

Expression	Parenthesized execution order
`not false = true`	`not (false = true)`
`a | b & c | d = 110`	`(a | (b & c) | d) = 110`
`~10 + 5 * 8`	`(~10) + (5 * 8)`
`1 / 2 * 10`	`(1 / 2) * 10`
`1 + 3 * 15 % 2 * 4`	`1 + ((3 * 15) % 2) * 4`

Please note that the precedence of operators vary between different database vendors. Use parenthesis to make sure the operations are executed as intended.

Standard Compliance

This section summarizes standard compliance concerning operators.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
	Mimer SQL extension	Support for `%` modulo operator is a Mimer SQL extension. Support for bit operators is a Mimer SQL extension.

Value Specifications

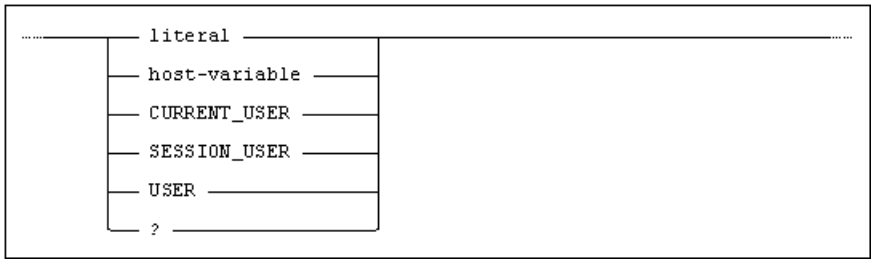
Specifying fixed values in expressions.

Value specifications are values which are fixed within the context of one SQL statement. Value specifications are different to values derived from column contents, which can change as different rows or sets of rows are addressed.

The value specifications which may be used in expressions are:

- literals, see *Literals* on page *64*
- parameter markers and host variables, see *Parameter Markers and Host Identifiers* on page *41*, and *Mimer SQL Programmer's Manual, Chapter 4, Using Host Variables* respectively.
- the keyword `CURRENT_USER`, `SESSION_USER` or `USER`, representing the name of a current ident (a national character varying string with maximum length 128). See *SESSION_USER* on page *124* and *CURRENT_USER* on page *99* respectively.

In the syntax diagrams, the term `value-specification` may be replaced by the following construction:



Standard Compliance

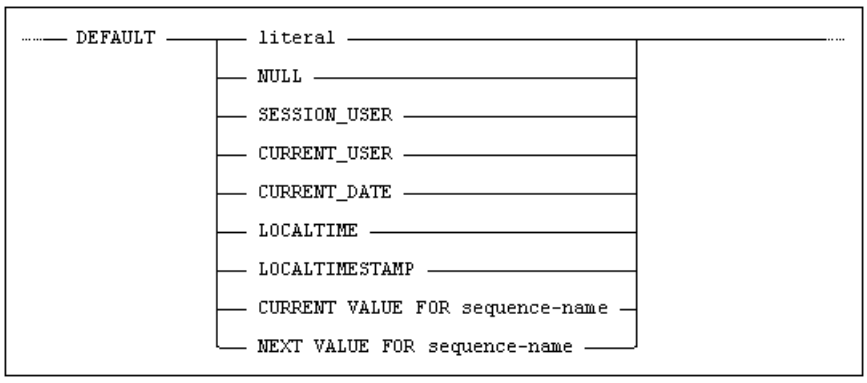
This section summarizes standard compliance for value specifications.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Feature outside core	Feature F561, “Full value expressions”.

Default Values

There are various places in the Mimer SQL syntax where a default value can be specified. The value resulting from a default value specification must always be assignment-compatible with the data type of the context to which it will be applied.

In the syntax diagrams, the term `default-value` may be replaced by the following construction:



For more information about what can be specified for `literal`, see *Literals* on page *64*.

Standard Compliance

This section summarizes standard compliance concerning the specification of default values.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F555, “Enhanced seconds precision”. Localtime and Localtimestamp functions with fractions of seconds. Feature T176, “Sequence generator support”
	Mimer SQL extension	Current value for sequences is a Mimer SQL extension.

Assignments

The following sections explain the rules that apply when values are assigned in SQL statements to database columns or to host variables.

String Assignments

If a string value assigned to a fixed-length or variable-length character column is longer than the defined length of the column (except for trailing spaces), the assignment will fail and an error is returned.

If a string value assigned to a fixed-length character column is shorter than the defined length of the column, the content of the column is padded to the right with blanks after the assignment.

If a string value assigned to a variable-length character column is shorter than the defined maximum length of the column, no blank padding occurs.

Character (both fixed length and variable length) column values assigned to fixed-length host variables in SQL statements are padded with blanks to the right if necessary. Column values assigned to host variables are truncated if they are longer than the declared length of the variable, and a warning is issued.

The following table summarizes the rules for character string assignment:

Assignment	Source too long	Source too short
To column	Error if non-blank character would be truncated.	Pad right with blanks for fixed-length columns. No blank padding for variable length columns.
To variable	Truncate and warn.	Pad right with blanks for fixed-length variables. No blank padding for variable length variables.

Numerical Assignments

Numbers assigned to columns or host variables assume the data type of the item to which they are assigned, regardless of the data type of the source.

Integral parts of INTEGER, DECIMAL or FLOAT values are never truncated. Fractional parts of DECIMAL and FLOAT numbers may be truncated if required. No precision is lost when converting INTEGER values to DECIMAL, but this may happen when converting INTEGER values to FLOAT.

When DECIMAL or FLOAT values are converted to INTEGER, the fractional part of the number is truncated (not rounded). Note that the range of numbers represented by DECIMAL and INTEGER is smaller than the range represented by FLOAT. Assignment of a FLOAT number to an INTEGER or DECIMAL produces an overflow error if the source number is too large.

In assigning DECIMAL values to DECIMAL targets, the length of the integer part of the source (i.e. the difference between the precision and scale) may not exceed the precision of the target. The necessary number of leading zeros is appended or eliminated, and trailing zeros are added to or digits truncated from the fractional part as required.

Note: Truncation effects can be avoided by explicitly using the ROUND function, see *ROUND* on page *123*.

In converting DECIMAL values to FLOAT, the mantissa of the target is treated as a decimal number with the same precision as the source (for example, 1234.56 becomes 1.23456E3).

In converting FLOAT values to DECIMAL, digits are truncated from the fractional part of the result as required by the scale of the target. An overflow error occurs if the precision of the target cannot accommodate the integral part of the result.

When converting INTEGER, DECIMAL or FLOAT numbers to REAL or DOUBLE PRECISION, a rounding operation is often required. The number will be rounded to the nearest binary floating point representation (rounding to even if there is a tie). Note that such rounding is necessary for simple decimal numbers such as 0.1 which cannot be represented exactly as a binary floating point number.

When converting a REAL or DOUBLE PRECISION number to INTEGER, DECIMAL or FLOAT, the value will be rounded to the nearest number that is possible to represent in the target.

The following table illustrates the main features of numerical assignments:

Source:	**Target:** INTEGER	SMALLINT	DECIMAL(9,2)	FLOAT(8)	REAL
INTEGER(6): 987654	987654	Overflow	987654.00	9.8765400E5	9.87654000E5
DECIMAL(6,3): 987.654	987	987	987.65	9.8765400E2	9.87654000E5
FLOAT(6): 9.87654E5	987654	Overflow	987654.00	9.8765400E5	9.87654000E5
FLOAT(6): 9.87654E49	Overflow	Overflow	Overflow	9.8765400E49	Overflow
FLOAT(6): 9.87654E-49	0	0	0.00	9.8765400E-49	0.0E0
REAL: 0.3E0	0	0	0.30	3.0000001E-001	3.00000012E-01

Leading zeros are shown where appropriate to indicate the maximum number of digits available. Leading zeros in numerical data are not normally displayed on output.

Datetime Assignment Rules

The following compatibility rules apply when assigning DATETIME values to one another:

- If the value to be assigned is a DATE, the target must also be a DATE.
- If the value to be assigned is a TIME, the target must also be a TIME.
- If the value to be assigned is a TIMESTAMP, the target must also be a TIMESTAMP.
- The CAST function can be used in order to cross-assign.

Interval Assignment Rules

The following compatibility rules apply when assigning INTERVAL values to one another:

- When assigning a non-null value to an INTERVAL column, the leading precision of the target must be sufficient to represent the value.
- All YEAR-MONTH INTERVAL values are compatible with one another.
- All DAY-TIME INTERVAL values are compatible with one another.

Binary Assignment Rules

A binary value assigned to a fixed-length binary column must have the same length as the defined length of the column, otherwise the assignment will fail and an error is returned.

If a binary value assigned to a variable-length binary column is shorter than the defined maximum length of the column, current length is set for the column.

If a binary value assigned to a variable-length binary column is longer than the defined maximum length of the column, the assignment will fail and an error is returned.

Binary (both fixed length and variable length) column values assigned to fixed-length host variables in SQL statements are padded with null values to the right if necessary. Column values assigned to host variables are truncated if they are longer than the declared length of the variable, and a warning is issued.

The following table summarizes the rules for binary string assignment:

Assignment	Source too long	Source too short
To column	Error.	Error for fixed-length columns. Current length set for variable length columns.
To variable	Truncate and warn.	Pad right with null values for fixed-length variables. No null value padding for variable length variables.

Boolean Assignment Rules

The BOOLEAN type can be assigned boolean values, i.e. TRUE and FALSE.

Note: Do not enclose boolean literals in string delimiters. 'TRUE' is a string literal, not a boolean literal.

Standard Compliance

This section summarizes standard compliance concerning assignments.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Comparisons

Values to be compared must be of compatible data types. If values with incompatible data types are compared, an error occurs.

Character String Comparisons

Both fixed-length and variable-length character strings are compared character by character from left to right.

If the strings are of different length, the shorter string is conceptually padded to the right with blanks before the comparison is made, that is, character differences take precedence over length differences.

For example, the variable-length column with the value '`town `', one trailing blank, is equal to the variable-length column with the value '`town  `', two trailing blanks.

When comparing a character string to a national character string, the character string is implicitly converted to a national character string, before the comparison is performed.

Collations

A collation determines whether a character string is less than, equal to, or greater than another when sorting or comparing data.

SQL only permits compatible character strings to be compared. That is, you can compare character strings only if the source and target strings belong to the same collation or are coerced into having the same collation.

A character string that is defined with a named collation can only be compared to a character string that is either defined with the same named collation or is defined without a collation.

In the case where one of the strings is not associated with a named collation then it will be implicitly coerced to the same collation as the other string.

A collation specified in the column-definition will take precedence over a domain collation.

For more information on character sets, see *Appendix B Character Sets*.

For more information on collations, see *Mimer SQL User's Manual, Chapter 4, Collations*.

Numerical Comparisons

INTEGER, DECIMAL and FLOAT values are always compared according to their algebraic values.

INTEGER values compared with DECIMAL or FLOAT values are treated as DECIMAL or FLOAT respectively. When DECIMAL values are compared with DECIMAL, the lower precision value is conceptually padded with leading and trailing zeros as necessary. DECIMAL values compared with FLOAT values are treated as FLOAT.

Thus all the following comparisons evaluate to TRUE:

```
1 = 1.0
2 < 2.3E0
35.3 = 035.300
35.3 > 3.5E1
```

A REAL, DOUBLE PRECISION or FLOAT value (A) compared with an INTEGER, DECIMAL or FLOAT value (B) is handled in the following manner:

```
A' = CAST(A AS type-of-B);
Q = (A = CAST(A' AS type-of-A));
```

If Q is TRUE then A' is a close approximation of A in he type of B. Comparisons are made between A' and B in the type B.

If Q is FALSE then A' is NOT a close approximation of A in the type of B. A is considered to be unequal to any value in type B. However A' and B can be compared for magnitude (> or <).

Thus all the following comparisons evaluate to TRUE:

```
CAST(1 AS REAL) = 1
CAST(1.1 AS REAL) <> 1
CAST(1.1 AS REAL) = 1.1
CAST(1.1 AS REAL) <> 1.10000000
CAST(1.1 AS REAL) = 1.10000002
```

Datetime and Interval Comparisons

Two `DATETIME` values may be compared if they are assignment-compatible, as defined in *Datetime Assignment Rules* on page *79*.

`DATETIME` comparisons are performed in accordance with chronological ordering.

When two `TIME` or two `TIMESTAMP` values are compared, the seconds precision of the value with the lowest seconds precision is extended by adding trailing zeros.

Two `INTERVAL` values may be compared if they are assignment-compatible, as defined in *Interval Assignment Rules* on page *79*.

`INTERVAL` comparisons are performed in accordance with their sign and magnitude.

It is not possible to compare `YEAR-MONTH` intervals with `DAY-TIME` intervals.

Comparable `INTERVAL` types with different interval precisions are conceptually converted to the same interval precision, prior to any comparison, by adding fields as required.

Binary Comparisons

Binary values are compared bytewise. If the two binary values have different lengths they are not equal.

Boolean Comparisons

Boolean values are compared to TRUE or FALSE. When comparing truth values FALSE is less than TRUE.

When equals true is to be evaluated it is unnecessary to write the = TRUE part. I.e.

```
WHERE boolcol = TRUE
```

is typically written as

```
WHERE boolcol
```

Similarly, = FALSE is typically re-written using NOT. I.e.

```
WHERE boolcol = FALSE
```

is usually expressed as

```
WHERE NOT boolcol
```

The BOOLEAN TEST syntax is supported for truth value tests, i.e.:

boolean-primary IS TRUE

boolean-primary IS FALSE

boolean-primary IS UNKNOWN

boolean-primary IS NOT TRUE

boolean-primary IS NOT FALSE

boolean-primary IS NOT UNKNOWN

Null Comparisons

All comparisons involving a null value on either side of the comparison operator evaluate to unknown. Null is never equal to, greater than or less than anything else.

SQL provides a special NULL predicate to test for the presence or absence of null value in a column, see *The NULL Predicate* on page *159*.

The DISTINCT predicate provides a comparison mechanism that treats two null values as the same, see *The DISTINCT Predicate* on page *161*.

Considerable care is required in writing search conditions involving columns which may contain null values. It is often very easy to overlook the effect of null comparisons, with the result that rows which should be included in the result table are omitted or vice versa. See the *Mimer SQL User's Manual, Chapter 3, Handling Null Values,* for further discussion of this point.

Truth Tables

The following truth tables summarize the outcome of conditional expressions where comparisons are negated by NOT or joined by AND or OR.

A question mark (?) represents the truth value unknown, T represents the value TRUE and F represents the value FALSE.

NOT

NOT	
T	F
F	T
?	?

AND

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

IS

IS	T	F	?
T	T	F	F
F	F	T	F
?	F	F	T

Standard Compliance

This section summarizes standard compliance concerning comparisons.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F571, “Truth value tests” Feature T031, “BOOLEAN data type”

Result Data Types

This section describes the syntax rules for, and the resulting data types of, `UNION`, `INTERSECT` and `EXCEPT` operations specified in a `query-expression` (see `SELECT`) and `CASE` expressions, see *CASE Expression* on page *145*.

- The data type of all specified expressions must be comparable.
- If any of the specified expressions is a variable-length (national) character string, then the data type of the result will be variable-length (national) character with maximum length equal to the largest of the specified expressions.
- If all specified expressions are fixed-length (national) character strings, then the data type of the result will be a fixed-length (national) character string with a length equal to the maximum length of the largest of the specified fixed-length character string values.
- If all specified expressions are boolean, then the data type of the result will be boolean.
- If any of the specified expressions is variable-length binary, then the data type of the result will be variable-length binary with maximum length equal to the maximum length of the largest of the specified expressions.
- If all specified expressions are fixed-length binary, then the data type of the result will be fixed-length binary with the same length.
- If all specified expressions are exact numeric, then the data type of the result will be exact numeric with precision and scale equal to the maximum precision and scale of the specified expressions.
- If any of the specified expressions is approximate numeric, then the data type of the result will be approximate numeric with precision equal to the maximum precision of the specified expressions.
- If two numeric data types are specified, the precision and scale of the result is determined by the rules in the table below and which are described in the points that follow:

	`FLOAT(p")`	`INTEGER(p")`	`DECIMAL(p",s")`
`FLOAT(p')`	`FLOAT(p)` [a]	`FLOAT(p)` [a]	`FLOAT(p)` [a]
`INTEGER(p')`	`FLOAT(p)` [a]	`INTEGER(p)` [a]	`DECIMAL(p,s)` [b]
`DECIMAL(p',s')`	`FLOAT(p)` [a]	`DECIMAL(p,s)` [b]	`DECIMAL(p,s)` [b]

a. p = max(p',p")
b. p = min(45, max(p'-s',p"-s")+max(s',s"))
s = max(s',s")

- If either of the specified expressions is floating point, the result is floating point. The precision of the result is the highest operand precision.

  Thus:

  `DOUBLE PRECISION UNION REAL` gives `DOUBLE PRECISION`

  `DOUBLE PRECISION UNION INTEGER` gives `DOUBLE PRECISION`

  `REAL UNION SMALLINT` gives `REAL`.

- If both the specified expressions are integer, the result is integer. The precision of the result is the highest operand precision.

  Thus:

  `INTEGER UNION SMALLINT` gives `INTEGER`

  `INTEGER UNION BIGINT` gives `BIGINT`.

- If both the specified expressions are decimal, or one is decimal and the other is integer, the result is decimal. For expressions mixing decimal and integer operands, `INTEGER(p)` is treated as `DECIMAL(p,0)`.

  The number of positions to the left of the decimal point (i.e. the difference between precision and scale) in the result is the greatest number of positions in either operand. The scale of the result is the greatest scale of the operands. The precision may not exceed 45.

  Thus:

  `SMALLINT UNION DECIMAL(10,4)` gives `DECIMAL(10,4)`

  `INTEGER UNION DECIMAL(10,4)` gives `DECIMAL(14,4)`

  `DECIMAL(9,2) UNION DECIMAL(6,4)` gives `DECIMAL(9,4)`.

- For `INTERVAL` operands, see *Interval* on page *54*, the interval precision of the result is the combined interval precision of the two operands, the scale (seconds precision) is the greatest of the two operands and the leading precision of the result is the greatest of the two operands, expressed in terms of the most significant field of the result.

  Thus:

  `DAY TO HOUR UNION MINUTE TO SECOND` gives `DAY TO SECOND`

  `HOUR TO SECOND(2,2) UNION MINUTE TO SECOND(1,6)` gives `HOUR TO SECOND(2,6)`

  `DAY(2) TO HOUR UNION HOUR(6) TO MINUTE` gives `DAY(5) TO MINUTE`.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Chapter 8

Functions

This chapter discusses scalar functions and set functions (see *Set Functions* on page *136*.)

Scalar Functions

A scalar function takes zero or more parameters and returns a single value. A scalar function can be used wherever an expression is allowed.

Scalar functions

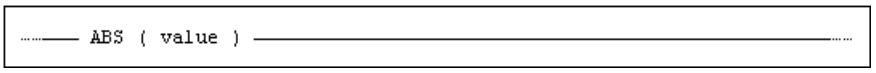
The following sections describe Mimer SQL's scalar functions.

ABS

Returns the absolute value of the given numeric expression.

Syntax

Syntax for the `ABS` function:



`value` is a numeric or an interval value expression.

Description

The function returns the absolute value of `value`.

If the value of value is null, then the result of the function is null.

Example

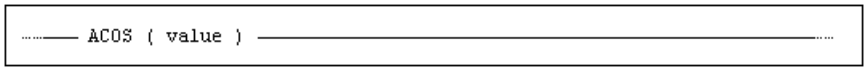
```
SET INT_VAL = ABS(-15);  -- sets INT_VAL to 15
```

ACOS

Returns the arccosine for a numeric expression.

Syntax

Syntax for the `ACOS` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

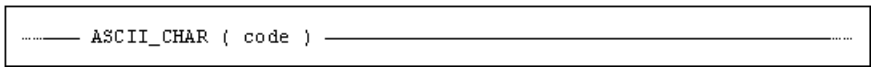
- The functions returns the arccosine for the value expressed as radians. The data type for the result is double precision. Valid input values are in the range -1 to 1.
- If the value of `value` is NULL, then the result of the function is NULL.

ASCII_CHAR

Returns the character that has the given ASCII code value. The given ASCII code value should be in the range 0-255.

Syntax

Syntax for the `ASCII_CHAR` function:



`code` is a numeric expression representing an ASCII value.

Description

If the value of `code` is between 0 and 255, the function returns a single character value, i.e. `CHAR(1)`, otherwise the function returns null. (For `code` values above 255, use the `UNICODE_CHAR` function instead. See *UNICODE_CHAR* on page *131*.)

If the value of `code` is null, then the result of the function is null.

Example

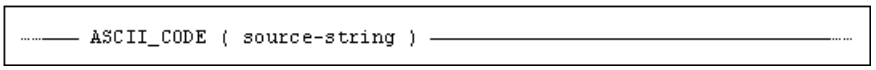
```
SET CHR_VAL = ASCII_CHAR(65);  -- sets CHR_VAL to 'A'
```

ASCII_CODE

Returns the ASCII code value of the leftmost character in the given string expression, as an integer.

Syntax

Syntax for the `ASCII_CODE` function:



`source-string` is a character or binary string expression.

Description

A single `INTEGER` value is returned, representing an ASCII code.

If the `source-string` contains more than one character, the ASCII code of the left-most octet is returned.

If the length of `source-string` is zero, then the result of the function is null.

If the value of `source-string` is null, then the result of the function is null.

Example

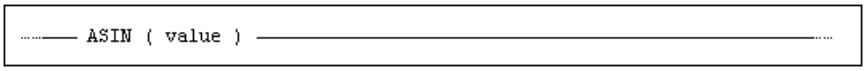
```
SET INT_VAL = ASCII_CODE('A');  -- sets INT_VAL to 65
```

ASIN

Returns the arcsine for a numeric expression.

Syntax

Syntax for the `ASIN` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

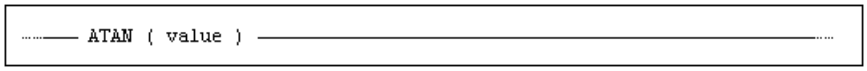
- The functions returns the arcsine for the value expressed as radians. The data type for the result is double precision. Valid input values are in the range -1 to 1.
- If the value of `value` is NULL, then the result of the function is NULL.

ATAN

Returns the arctangent for a numeric expression.

Syntax

Syntax for the ATAN function:



value is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

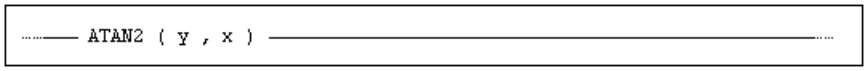
- The functions returns the arctangent for the value expressed as radians. The data type for the result is double precision.
- If the value of value is NULL, then the result of the function is NULL.

ATAN2

Returns the arctangent for the tangent between 2 numeric expressions.

Syntax

Syntax for the ATAN2 function:



y and x are numeric value expressions. The function handles values that are within the range of a double precision expression.

The ATAN2 function calculates the arctangent of the two parameters y and x. It is similar to calculating the arctangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result. Effectively, this means that ATAN2(y, x) finds the counterclockwise angle in radians between the x-axis and the vector <x, y> in 2-dimensional Euclidean space.

Rules

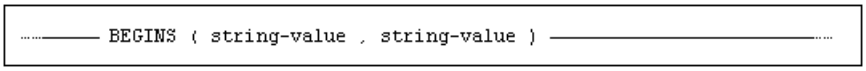
- Returns the angle, in radians, between the x-axis and the vector <x, y>. The data type for the result is double precision.
- If the value of y or x is NULL, then the result of the function is NULL.
- ATAN2(0.0, 0.0) raises an error, because a null vector is a point and does not have an angle.

BEGINS

Perform a “begins with” comparison.

Syntax

Syntax for the `BEGINS` function:



Description

`LIKE` predicates, addressing the “begins with” functionality, are very common.

However, when a parameter marker is used for the LIKE pattern, the SQL compiler can not determine the LIKE pattern characteristics, and possible optimizations will not be applied. The built-in function `BEGINS` will overcome this issue.

Examples

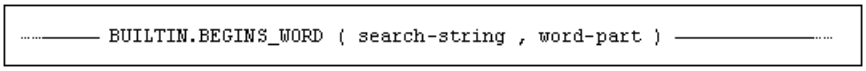
BEGINS function	Is equivalent to
`BEGINS(col,'AB')`	`col LIKE 'AB%'`
`BEGINS(col,?),` `where ? contains 'XYZ'`	`col LIKE 'XYZ%'`

BUILTIN.BEGINS_WORD

Returns a boolean denoting if there is a word in the `search-string` argument that begins with the `word-part` argument.

Syntax

Syntax for the `BUILTIN.BEGINS_WORD` function:



Description

The `search-string` and the `word-part` arguments must both be character expressions (i.e. either CHARACTER/VARCHAR or NATIONAL CHARACTER/NVARCHAR.)

For this type of searches, the database will consider using a `WORD_SEARCH` index if appropriate. (See *CREATE INDEX* on page *264*.)

If any of the arguments to the function is null the function returns null. The function will return true if there is a word in the `search-string` argument that begins with the characters in the `word-part` argument and false otherwise.

Trailing space characters in the `<word-part>` string are trimmed before the search operation. The `<word-part>` string may only contain characters that have the Unicode property "ID_Continue". For a detailed description, see https://www.unicode.org/reports/tr31.

Examples

```
SQL>set ? = builtin.begins_word('The quick brown fox jumps over', 'bro');
?
=====
TRUE
```

The following comparison will not match since the case of the `word-part` does not match.

```
SQL>set ? = builtin.begins_word('The quick brown fox jumps over','Bro');
?
=====
FALSE
```

It is possible to use collations for the arguments, for example to do a case insensitive search:

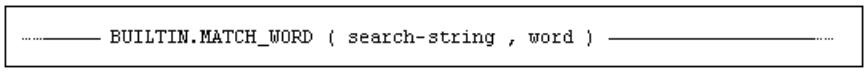
```
SQL>set ? = builtin.begins_word('The quick brown fox jumps',
SQL&'Bro' collate english_1);
?
====
TRUE
```

BUILTIN.MATCH_WORD

Returns a boolean denoting if there is a word in the `search-string` that matches the `word` argument.

Syntax

Syntax for the `BUILTIN.MATCH_WORD` function:



Description

The `search-string` and the `word` arguments must both be character expressions, either character or national character.

For this type of searches, the database will consider using a `WORD_SEARCH` index if appropriate. (See *CREATE INDEX* on page *264*.)

If any of the arguments to the function is null the function returns null. The function will return true if there is a word in the `search-string` argument that matches the `word` argument completely, and false otherwise.

Trailing space characters in the `<word>` string are trimmed before the match operation. The `<word>` string may only contain characters that have the Unicode property "ID_Continue". For a detailed description, see https://www.unicode.org/reports/tr31.

Examples

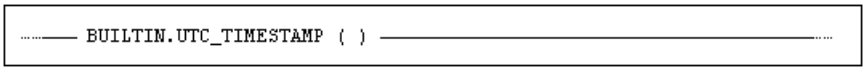
```
SQL>set ? = builtin.match_word('The quick brown fox jumped', 'bro');
?
=====
FALSE

SQL>set ? = builtin.match_word('The quick brown fox jumped', 'brown');
?
=====
TRUE

SQL>create index docind on ducuments (content for word_search);
SQL>select * from documents where builtin.match_word(content, 'Mimer');
```

BUILTIN.UTC_TIMESTAMP

Returns a timestamp denoting the current Coordinated Universal Time.

Syntax

Syntax for the `BUILTIN.UTC_TIMESTAMP` function:



Description

The result is the current Coordinated Universal Time as a timestamp value.

All references to `BUILTIN.UTC_TIMESTAMP` are effectively evaluated simultaneously from a single reading of the server clock. Thus the conditional expression `BUILTIN.UTC_TIMESTAMP() = BUILTIN.UTC_TIMESTAMP()` is guaranteed to always evaluate to true.

Examples

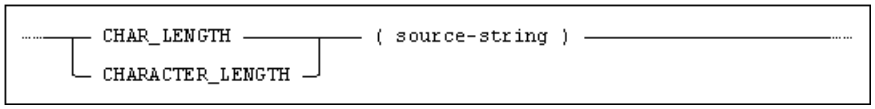
```
SQL>SELECT BUILTIN.UTC_TIMESTAMP() AS utcts FROM system.onerow;

utcts
=====
2012-10-30 14:55:22.643082

One row found


CREATE TABLE EVENTS(ID INTEGER PRIMARY KEY,
                    UTCTS TIMESTAMP);
INSERT INTO EVENTS(ID) VALUES (1, BUILTIN.UTC_TIMESTAMP());
UPDATE EVENTS
  SET    ID = ID + 5, UTCTS = BUILTIN.UTC_TIMESTAMP()
  WHERE  ID = 10;
```

CHARACTER_LENGTH

Returns the length of a string.

Syntax

Syntax for the CHARACTER_LENGTH (or CHAR_LENGTH) function:



source-string is a character or binary string expression.

Description

CHAR_LENGTH returns an INTEGER value.

If the data type of source-string is variable-length character or variable-length binary, then the result of CHAR_LENGTH is the same as the actual length of source-string.

If the data type of source-string is fixed-length character or fixed-length binary, then the result of CHAR_LENGTH is the same as the fixed-length of source-string.

If the value of source-string is null, then the result of the function is null.

Example

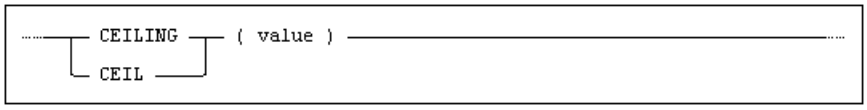
```
SET INT_VAL = CHAR_LENGTH('TEST STRING');  -- sets INT_VAL to 11
```

CEILING

Returns the smallest integer greater than or equal to a numeric expression.

Syntax

Syntax for the CEILING function:



value is a numeric value expression.

Description

The function returns the nearest integer value that is equal or higher to value.

If the value of value is null, then the result of the function is null.

The return data type is based on the input data type. For DECIMAL input, the return data type is integer.

Example

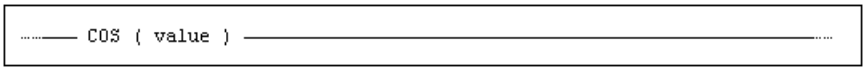
```
SET ? = CEILING(3.57);      -- returns 4
SET ? = CEILING(-3.57);     -- returns -3
SET ? = CEILING(1.2345e3); -- returns 1.235000000E+003
```

COS

Returns the cosine for a numeric expression.

Syntax

Syntax for the `COS` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

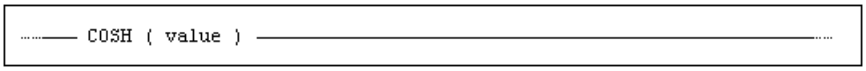
- The functions returns the cosine for the values expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

COSH

Returns the for hyperbolic cosine a numeric expression.

Syntax

Syntax for the `COSH` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

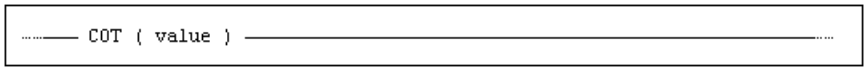
- The functions returns the hyperbolic cosine for the values expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

COT

Returns the cotangent for a numeric expression.

Syntax

Syntax for the `COT` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

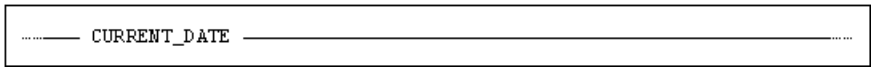
- The function returns the cotangent for the value, expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

CURRENT_DATE

Returns a `DATE` value denoting the current date (i.e. today).

Syntax

Syntax for the `CURRENT_DATE` function:



Description

The result is the current date (i.e. today) as a `DATE` value.

All references to `CURRENT_DATE` are effectively evaluated simultaneously from a single reading of the server clock. Thus the conditional expression `CURRENT_DATE = CURRENT_DATE` is guaranteed to always evaluate to true.

The value of `CURRENT_DATE` will always be equal to the `DATE` portion of `LOCALTIMESTAMP`.

Example

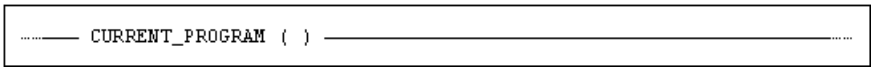
```
UPDATE sometable SET usercnt = 13, updated = CURRENT_DATE;
```

CURRENT_PROGRAM

Returns the name of an entered program.

Syntax

Syntax for the CURRENT_PROGRAM function:



Description

The function returns the value of the most recently entered program as nchar varying value with a maximum length of 128, with the collation SQL_IDENTIFIER.

If no program has been entered the result of the function is null.

Example

The following example returns the PROGRAM ident if entered, otherwise the session ident:

```
SET CHR_STR = COALESCE(CURRENT_PROGRAM(), SESSION_USER);
```

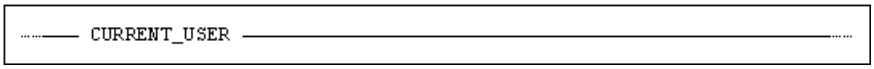
CURRENT_USER

Returns the name of the currently connected USER ident or the PROGRAM ident that is currently entered.

When used in a routine or trigger, it returns the name of the creator of the schema to which the routine or trigger belongs.

Syntax

Syntax for the CURRENT_USER function:



Description

When used in a routine or trigger, the result is the name of the creator of the schema to which the routine or trigger belongs, otherwise the value is the name of the connected ident or the program that was entered.

The data type of the returned value is nchar varying with a maximum length of 128, with the collation SQL_IDENTIFIER.

Example

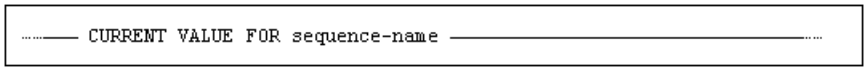
```
CREATE DOMAIN NAME AS NCHAR VARYING(128) collate SQL_IDENTIFIER DEFAULT
CURRENT_USER;
```

CURRENT VALUE

Returns the current value of a sequence.

Syntax

Syntax for the CURRENT VALUE function:



Description

The result is the current value of the sequence specified in sequence-name. This is the value that was returned when the NEXT VALUE function was used for this sequence in this session.

This function can not be used until the initial value has been established for the sequence by using NEXT VALUE (i.e. using it immediately after the sequence has been created will raise an error).

The function can be used where a value-expression would normally be used. It can also be used after the DEFAULT clause in the CREATE DOMAIN, CREATE TABLE and ALTER TABLE statements.

USAGE privilege must be held on the sequence in order to use it.

Example

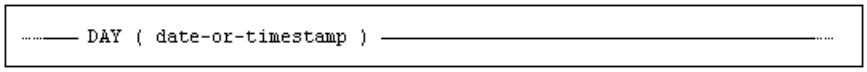
```
CREATE DOMAIN CHARGE_PERIOD_VALUE AS INTEGER
DEFAULT CURRENT VALUE FOR CHARGE_PERIOD_NO_SEQUENCE;
```

DAY

Returns the day of the month for the given date expression, expressed as an integer value in the range 1-31.

Syntax

Syntax for the DAY function:



date-or-timestamp is a date or timestamp value expression.

Description

The result is an integer value, 1 through 31.

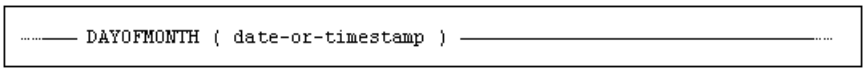
If the value of date-or-timestamp is null, then the result of the function is null.

DAYOFMONTH

Returns the day of the month for the given date expression, expressed as an integer value in the range 1-31.

Syntax

Syntax for the `DAYOFMONTH` function:



`date-or-timestamp` is a date or timestamp value expression.

Description

The result is an integer value, 1 through 31.

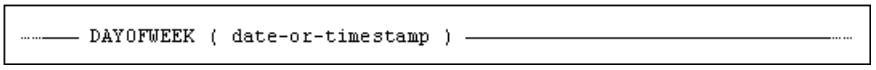
If the value of `date-or-timestamp` is null, then the result of the function is null.

DAYOFWEEK

Returns the day of the week for the given date expression, expressed as an integer in the range 1-7.

Syntax

Syntax for the `DAYOFWEEK` function:



`date-or-timestamp` is a date or timestamp value expression.

Description

The result is an integer value, 1 through 7, where `1` = Monday.

If the value of `date-or-timestamp` is null, then the result of the function is null.

Example

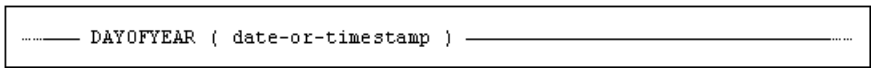
```
SET INT_VAL = DAYOFWEEK(CURRENT_DATE);  -- sets INT_VAL to the
                                        -- day number of the current week

SET INT_VAL = DAYOFWEEK(DATE'2024-12-03');  -- sets INT_VAL to 2, for Tuesday
```

DAYOFYEAR

Returns the day of the year for the given date expression, expressed as an integer in the range 1-366.

Syntax

Syntax for the `DAYOFYEAR` function:



`date-or-timestamp` is a date or timestamp value expression.

Description

The result is an integer value, 1 through 366, where `1` = January 1.

The value for a day after February 28 depends on whether the year is a leap year or not.

If the value of `date-or-timestamp` is null, then the result of the function is null.

Example

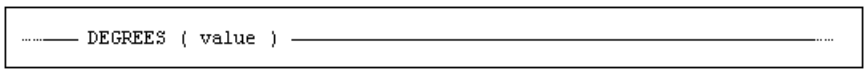
```
SET INT_VAL = DAYOFYEAR(CURRENT_DATE);  -- sets INT_VAL to the
                                        -- day number of the current year

SET INT_VAL = DAYOFYEAR(DATE'2016-11-10');  -- sets INT_VAL to 315
SET INT_VAL = DAYOFYEAR(DATE'2017-11-10');  -- sets INT_VAL to 314
```

DEGREES

Returns an angle expressed in radians as degrees.

Syntax

Syntax for the `DEGREES` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

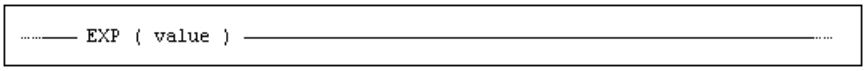
- The function converts a numeric expression in radians to the corresponding values expressed in degrees. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

EXP

Returns the exponential value for a numeric expression.

Syntax

Syntax for the `EXP` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

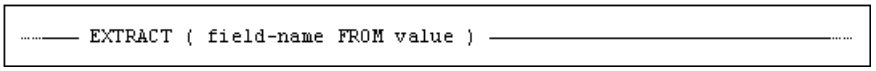
- The function returns exponential value for the value expression. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

EXTRACT

Extracts a single field from a `DATETIME` or `INTERVAL` value.

Syntax

Syntax for the `EXTRACT` function:



Description

`field-name` is one of: `YEAR`, `MONTH`, `DAY`, `HOUR`, `MINUTE` or `SECOND`.

`value` must be of type `DATETIME` or `INTERVAL` and it must contain the field specified by `field-name`, otherwise an error is raised.

The data type of the result is integer.

The exception is when `field-name` is `SECOND`, in which case the result type is decimal where the precision is equal to the sum of the leading precision and the seconds precision of `value`, with a scale equal to the seconds precision.

When `value` is a negative `INTERVAL`, the result is a negative value. In all other cases the result is a positive value.

If the value of `value` is null, then the result of the function is null.

Example

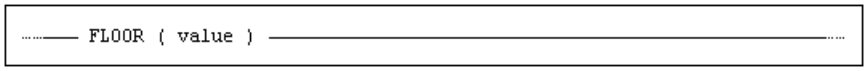
```
SELECT CASE EXTRACT (MONTH FROM ARRIVE)
         WHEN 1 THEN 'JANUARY'
         WHEN 2 THEN 'FEBRUARY'
         ....
       END
FROM TRAVELS
```

FLOOR

Returns the largest integer less than or equal to a numeric expression.

Syntax

Syntax for the `FLOOR` function:



`value` is a numeric value expression.

Description

The function returns the nearest integer value that is equal or lower to `value`.

If the value of `value` is null, then the result of the function is null.

The return data type is based on the input data type. For DECIMAL input, the return data type is integer.

Example

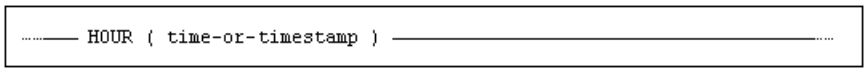
```
SET ? = FLOOR(13.13);     -- returns 13
SET ? = FLOOR(-13.13);    -- returns -14
SET ? = FLOOR(-12.34E1); -- returns -1.240000000E+002
```

HOUR

Returns the hour for the given time or timestamp expression, expressed as an integer value in the range 0-23.

Syntax

Syntax for the `HOUR` function:



`time-or-timestamp` is a time or timestamp value expression.

Description

The result is an integer value, 0 through 23, representing the hour.

If the value of `time-or-timestamp` is null, then the result of the function is null.

Example

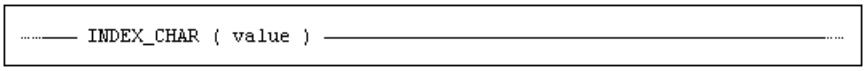
```
SET H = HOUR(LOCALTIME);  -- sets H to the current hour number
```

INDEX_CHAR

Returns the index character for a string.

Syntax

Syntax for the `INDEX_CHAR` function:



`value` is a character value expression

Description

The result is a character value.

If the value of `value` is null, then the result of the function is null.

The `INDEX_CHAR` function takes a character string as argument and returns the index character for the string related to its collation. The default behavior is to return the first letter of the string, decomposed (accents removed) and capitalized (upper case).

However, many languages include accented letters, digraphs, and sometimes trigraphs as basic alphabetical characters. These combinations are properly handled by the `INDEX_CHAR` function.

Examples

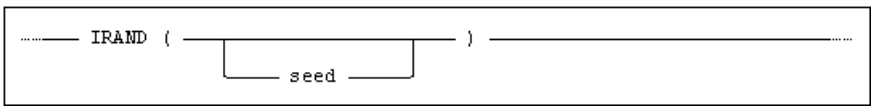
```
SELECT INDEX_CHAR('östra aros' COLLATE english_1) FROM... -- will return 'O'
SELECT INDEX_CHAR('östra aros' COLLATE swedish_1) FROM... -- will return 'Ö'
```

IRAND

Returns a random integer number.

Syntax

Syntax for the `IRAND` function:



`seed` is an integer value expression

Description

The result is a random integer value, in the range 0 to 2 147 483 647.

If a `seed` is given, this value is used to calculate the random value. If no `seed` is given, the value is calculated from the previous value. It is thus possible to generate the same random sequence by using the same `seed`.

Example

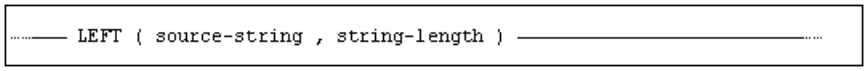
```
SET INT_VAL = MOD(IRAND(), 5);  -- sets INT_VAL to a random
                                -- value between 0 and 4
```

LEFT

Returns the specified number of leftmost characters in a given character string.

Syntax

Syntax for the `LEFT` function:



`source-string` is a character or binary string expression.

`string-length` is an integer value expressions.

Description

The leftmost `string-length` characters of `source-string` are returned.

If `count` is zero, an empty string is returned.

If `count` is less than zero, then the result of the function is null.

If the value of either operand is null, then the result of the function is null.

Example

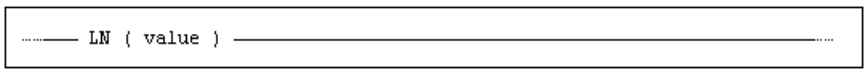
```
SET CHR_STR = LEFT('TEST STRING', 3);  -- sets CHR_STR to 'TES'
```

LN

Returns the natural logarithm for a numeric expression.

Syntax

Syntax for the `LN` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

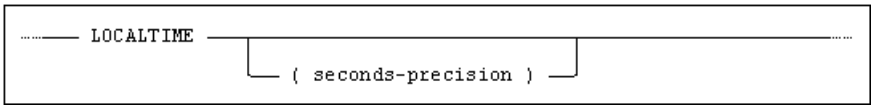
- The function returns the natural logarithm for the value expression. The data type for the result is double precision. Valid input values are > 0.
- If the value of `value` is NULL, then the result of the function is NULL.

LOCALTIME

Returns a `TIME` value denoting the current time (i.e. now).

Syntax

Syntax for the `LOCALTIME` function:



`seconds-precision` is an unsigned integer value denoting the seconds precision for the returned `TIME` value.

Description

The result is the current time (i.e. now) as a `TIME` value.

The value of `seconds-precision` must be between 0 and 9.

If `seconds-precision` is not specified, the default value of 0 is assumed.

All references to `LOCALTIME` are effectively evaluated simultaneously from a single reading of the server clock. Thus the conditional expression `LOCALTIME = LOCALTIME` is guaranteed to always evaluate to true.

The value of `LOCALTIME` will always be equal to the `TIME` portion of `LOCALTIMESTAMP`.

Example

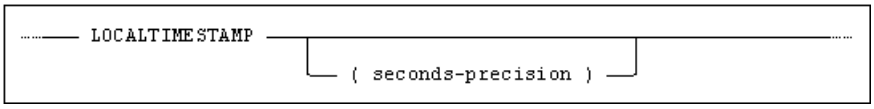
```
UPDATE EVENTS SET ADJUSTED = LOCALTIME   -- sets ADJUSTED to current time
WHERE ID = 81;                           -- (e.g. 15:45:02)
```

LOCALTIMESTAMP

Returns a `TIMESTAMP` denoting the current date and time.

Syntax

Syntax for the `LOCALTIMESTAMP` function:



`seconds-precision` is an unsigned integer value denoting the seconds precision for the returned `TIMESTAMP` value.

Description

The result is the current date and time as a `TIMESTAMP` value.

The value of `seconds-precision` must be between 0 and 9.

If `seconds-precision` is not specified, the default value of 6 is assumed.

All references to `LOCALTIMESTAMP` are effectively evaluated simultaneously from a single reading of the server clock. Thus the conditional expression `LOCALTIMESTAMP = LOCALTIMESTAMP` is guaranteed to always evaluate to true.

The value of `LOCALTIMESTAMP` will always be equal to the combined value of `CURRENT_DATE` and `LOCALTIME`.

Example

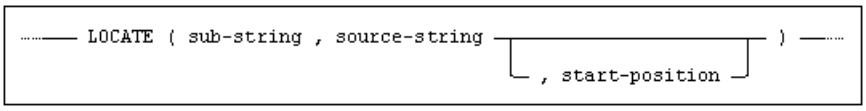
```
CREATE TABLE EVENTS(ID  INTEGER    PRIMARY KEY,
                    TS  TIMESTAMP  DEFAULT LOCALTIMESTAMP);
INSERT INTO EVENTS(ID) VALUES (1);  -- default value for TS inserted
                                    -- (e.g. 2019-09-27 16:14:07.230000)
UPDATE EVENTS
SET    TS = LOCALTIMESTAMP
WHERE  ID <= 10;
```

LOCATE

Returns the starting position of the first occurrence of a specified string expression in a given character string, starting from an optional start position, or the left of the character string.

Syntax

Syntax for the `LOCATE` function:



`sub-string` and `source-string` are character or binary string expressions.

`start-position` is an integer value expression.

Description

The position of the first occurrence of `sub-string` in `source-string` is returned, starting from the position specified by `start-position` if given, otherwise from position 1, in `source-string` (the left-most position).

If `sub-string` does not occur in `source-string`, the functions returns zero.

If the length of `source-string` is zero, the function returns zero.

If the length of `source-string` is less than `start-position`, the function returns zero.

If the length of `sub-string` is zero, the function returns 1.

If the value of any operand is null, then the result of the function is null.

Example

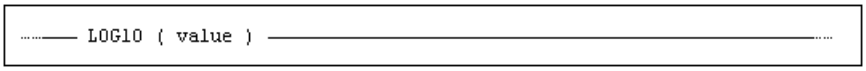
```
SET INT_VAL = LOCATE('NA', 'BANANA', 4);  -- sets INT_VAL to 5
```

LOG10

Returns the base-10 logarithm for a numeric expression.

Syntax

Syntax for the `LOG10` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

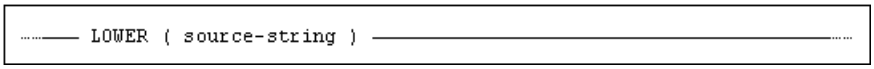
- The function returns the base-10 logarithm for the value expression. The data type for the result is double precision. Valid input values are > 0.
- If the value of `value` is NULL, then the result of the function is NULL.

LOWER

Converts all uppercase letters in a character string to lowercase.

Syntax

Syntax for the `LOWER` function:



`source-string` is a character string expression.

Description

The data type of the result is the same as the data type of `source-string`.

`source-string` is either in character or national character (i.e. Unicode) format.

If the value of `source-string` is null, then the result of the function is null.

Note: The length of the result may be longer or shorter than the input value. This means that using `LOWER` (or `UPPER`) on a column may cause data truncation.

Example

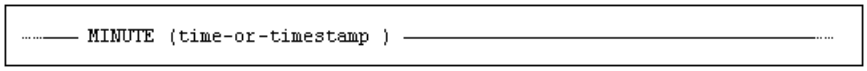
```
SELECT CHAR_LENGTH(TRIM(DESCRIPTION)), LOWER(TRIM(DESCRIPTION))
FROM   CHARGES;
```

MINUTE

Returns the minute for the given time or timestamp expression, expressed as an integer value in the range 0-59.

Syntax

Syntax for the `MINUTE` function:



`time-or-timestamp` is a time or timestamp value expression.

Description

The result is an integer value, 0 through 59, representing the minute.

If the value of `time-or-timestamp` is null, then the result of the function is null.

Example

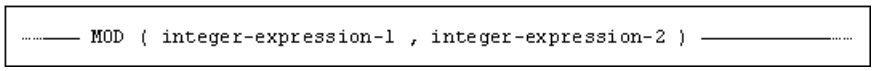
```
SET M = MINUTE(LOCALTIME);  -- sets M to the current minute number
```

MOD

Returns the remainder (modulus) of a specified integer expression divided by a second specified integer expression.

Syntax

Syntax for the `MOD` function:



`integer-expression-1` and `integer-expression-2` are integer value expressions.

Description

The result is the remainder of `integer-expression-1` divided by `integer-expression-2`.

If the value of `integer-expression-2` is zero, a divide-by-zero error will be raised.

The sign of the result is the same as the sign of `integer-expression-1`.

If the value of either operand is null, then the result of the function is null.

Note: Mimer SQL also supports the non-standard `%` modulo operator.

Example

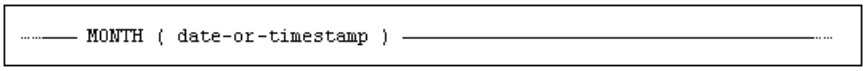
```
SET INT_VAL = MOD(IRAND(), 5);  -- sets INT_VAL to a random
                                -- value between 0 and 4
SET INT_VAL = IRAND() % 5;      -- sets INT_VAL to a random
                                -- value between 0 and 4
```

MONTH

Returns the month for the given date or timestamp expression, expressed as an integer value in the range 1-12.

Syntax

Syntax for the `MONTH` function:



`date-or-timestamp` is a date or timestamp value expression.

Description

The result is an integer value, 1 through 12, representing the month.

If the value of `date-or-timestamp` is null, then the result of the function is null.

Example

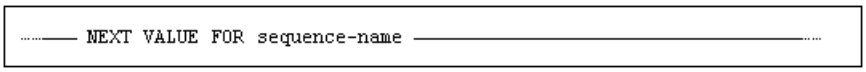
```
SET M = MONTH(CURRENT_DATE);  -- sets M to the current month number
```

NEXT VALUE

Returns the next value in the series of values defined by a sequence, provided that the last value in that series has not already been reached.

Syntax

Syntax for the `NEXT VALUE` function:



Description

The result will be the next value in the series of the values defined by the sequence specified in `sequence-name` (this value will then become the session' current value for the sequence).

If the sequence is unique (i.e. `NO CYCLE` option) and the current value of the sequence specified in `sequence-name` is already equal to the last value in the series of the values defined by it an error will be raised and the current value of the sequence will remain unchanged.

If the sequence is non-unique, the function will always succeed. If the current value of the sequence specified in `sequence-name` is equal to the last value in the series of values generated by the sequence, the initial value of the sequence will be returned.

The function can be used where a value-expression would normally be used. It can also be used after the `DEFAULT` clause in the `CREATE DOMAIN`, `CREATE TABLE` and `ALTER TABLE` statements.

This function is used to establish the initial value of the sequence after it has been created using the CREATE SEQUENCE statement.

USAGE privilege must be held on the sequence in order to use it.

Note: If the NEXT VALUE function is used in a select clause the sequence will be incremented for each row returned by the query.

Example

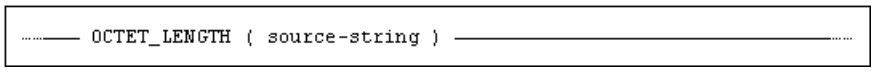
```
SET Z = NEXT VALUE FOR Z_SEQUENCE;
```

OCTET_LENGTH

Returns the octet (byte) length of a string. For single-octet character sets this is the same as CHARACTER_LENGTH.

Syntax

Syntax for the OCTET_LENGTH function:



source-string is a character or binary string expression.

Description

OCTET_LENGTH returns an INTEGER value.

If the data type of source-string is variable-length character or variable length binary, then the result of OCTET_LENGTH is the same as the actual length of source-string in octets.

If the data type of source-string is fixed-length character or fixed-length binary, then the result of OCTET_LENGTH is the same as the fixed-length of source-string.

If the data type of source-string is variable-length national character, then the result of OCTET_LENGTH is the same as the actual length of source-string in octets, i.e. 4 times the actual number of characters.

If the data type of source-string is fixed-length national character, then the result of OCTET_LENGTH is the same as 4 times the fixed-length of source-string.

If the value of source-string is null, then the result of the function is null.

Example

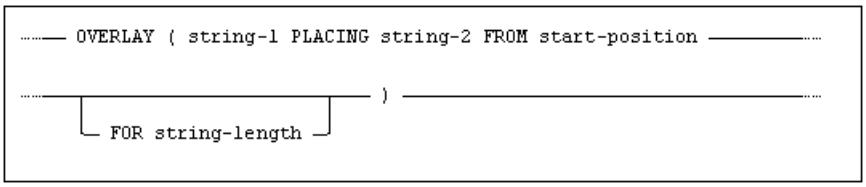
```
SET INT_VAL = OCTET_LENGTH(X'4142');  -- sets INT_VAL to 2
SET INT_VAL = OCTET_LENGTH('ABC');    -- sets INT_VAL to 3
SET INT_VAL = OCTET_LENGTH(n'ABC');   -- sets INT_VAL to 12
```

OVERLAY

Returns a character string where a number of characters, beginning at a given position, have been deleted from a character string and replaced with a given string expression.

Syntax

Syntax for the `OVERLAY` function:



`string-1` and `string-2` are character or binary string expressions.

`string-1` and `string-2` must be of the same type, i.e. either both character or both binary.

`start-position` and `string-length` are integer value expressions.

Description

The `string-length` number of characters in `string-1`, starting from position `start-position` are deleted from `string-1`. Then `string-2` is inserted into `string-1`, at the 'point of deletion'. The resulting character or binary string is returned.

If the value of `string-length` is positive, the `string-length` number of characters to the right of `start-position` are deleted. If the value of `string-length` is negative, the `string-length` number of characters to the left of `start-position` are deleted.

The point-of-deletion is where the cursor would be if you had just used a text editor to select the characters, as described, and performed an edit-cut operation.

A value for `start-position` of less than 1 (zero or negative) specifies a position to the left of the beginning of `string-1`.

It is possible that the specified deletion may not actually affect any of the characters of `string-1`, in which case the `OVERLAY` operation produces the effect of a prepend.

If the value of any operand is null, then the result of the function is null.

`string-2` must not contain Unicode characters outside the Latin1 repertoire if `string-1` is of character type.

Example

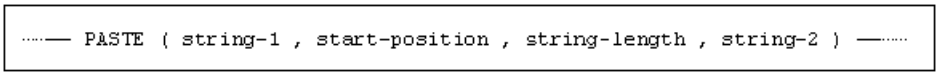
```
OVERLAY('ABCDEF' PLACING 'ab' FROM 2 FOR 3);  -- returns 'AabEF'
OVERLAY('ABCDEF' PLACING 'ab' FROM 2);        -- returns 'AabDEF'
```

PASTE

Returns a character string where a specified number of characters, beginning at a given position, have been deleted from a character string and replaced with a given string expression.

Syntax

Syntax for the PASTE function:



string-1 and string-2 are character or binary string expressions.

string-1 and string-2 must be of the same type, i.e. either both character or both binary.

start-position and string-length are integer value expressions.

Description

The string-length number of characters in string-1, starting from position start-position are deleted from string-1. Then string-2 is inserted into string-1, at the 'point of deletion'. The resulting character or binary string is returned.

If the value of string-length is positive, the string-length number of characters to the right of start-position are deleted. If the value of string-length is negative, the string-length number of characters to the left of start-position are deleted.

The point-of-deletion is where the cursor would be if you had just used a text editor to select the characters, as described, and performed an edit-cut operation.

A value for start-position of less than 1 (zero or negative) specifies a position to the left of the beginning of string-1.

It is possible that the specified deletion may not actually affect any of the characters of string-1, in which case the PASTE operation produces the effect of a prepend.

If the value of any operand is null, then the result of the function is null.

string-2 must not contain Unicode characters outside the Latin1 repertoire if string-1 is of character type.

Example

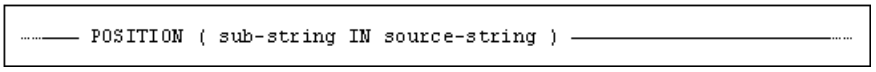
```
SET CHR_STR = PASTE('TEST STRING', 6, 3, 'P');  -- sets CHR_STR to 'TEST PING'
```

POSITION

Returns the starting position of the first occurrence of a specified string expression in a given character string, starting from the left of the character string.

Syntax

Syntax for the `POSITION` function:



`sub-string` and `source-string` are character or binary string expressions.

`sub-string` and `source-string` must be of the same type, i.e. either both character or both binary.

Description

The position of the first occurrence of `sub-string` in `source-string` is returned, starting from position 1 in `source-string` (the left-most position).

If `sub-string` does not occur in `source-string`, the functions returns zero.

If the length of `source-string` is zero, the function returns zero.

If the length of `sub-string` is zero, the function returns 1.

If the value of either operand is null, then the result of the function is null.

Example

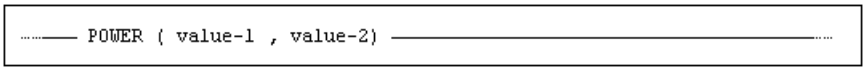
```
SET INT_VAL = POSITION('STR' IN 'TEST STRING');  -- sets INT_VAL to 6
```

POWER

Returns the specified numeric expression, raised to the power of the given value.

Syntax

Syntax for the `POWER` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

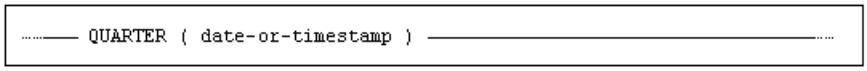
- The function returns the value of the first argument raised to the power of the second argument. The data type for the result is double precision.
- If the value of `value-1` or `value-2` is NULL, then the result of the function is NULL.

QUARTER

Returns the quarter for the given date or timestamp expression, expressed as an integer value in the range 1-4.

Syntax

Syntax for the `QUARTER` function:



`date-or-timestamp` is a date or timestamp value expression.

Description

The result is an integer value, 1 through 4, representing the quarter.

If the value of `date-or-timestamp` is null, then the result of the function is null.

Example

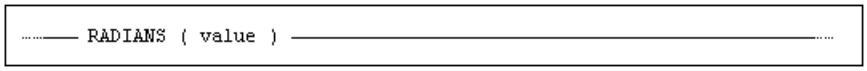
```
SET Q = QUARTER(CURRENT_DATE);  -- sets Q to the current quarter number
```

RADIANS

Returns an angle expressed in degrees as radians.

Syntax

Syntax for the `RADIANS` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

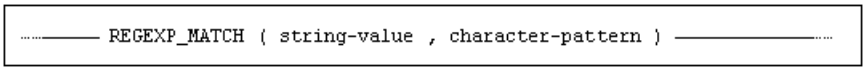
- The function converts a value expressed in degrees to the corresponding value expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

REGEXP_MATCH

Performs a regular expression comparison.

Syntax

Syntax for the `REGEXP_MATCH` function:



Description

The `REGEXP_MATCH` function compares the value in a string expression with a character string pattern which may contain different meta-characters.

Compared to `LIKE`, the regular expression provides a much more flexible way to match strings of text, such as complex patterns of characters.

Regular expression constructs

Characters

`x`	The character x
`\`	Escape for meta characters: $ & ( ) * + , - . ? [ ] ^ { | }
`\\`	The backslash character
`\t`	The tab character
`\n`	The newline character
`\v`	The vertical tab character
`\f`	The form feed character
`\r`	The carriage return character
`\x{h...h}`	The character with hex value 0xh...h (<= 0x10FFFF)

Character classes

`[abc]`	a, b, or c (simple class)
`[^abc]`	Any character except a, b, or c (negation)
`[a-zA-Z]`	a through z or A through Z, inclusive (range)
`[[a-d][m-p]]`	a through d, or m through p (union)
`[[a-z]&&[def]]`	d, e, or f (intersect)
`[[a-z]--[bc]]`	a through z, except for b and c (minus)

Predefined character classes	
`.`	Any character
`\d`	A digit character
`\D`	Not a digit character ([^\d])
`\s`	A whitespace character
`\S`	Not a whitespace character ([^\s])
`\w`	A word character
`\W`	Not a word character ([^\w])

Boundary matchers	
`^`	The beginning of string
`$`	The end of string

Quantifiers	
`X?`	X, once or not at all
`X*`	X, zero or more times
`X+`	X, one or more times
`X{n}`	X, exactly n times
`X{n,}`	X, at least n times
`X{n,m}`	X, at least n but not more than m times

Logical operators	
`XY`	X followed by Y
`X|Y`	Either X or Y
`(X)`	X, as a capturing group

Classes for Unicode categories	
`\p{L}`	Letter
`\p{Ll}`	Lowercase_Letter
`\p{Lu}`	Uppercase_Letter
`\p{Lt}`	Titlecase_Letter
`\p{Lm}`	Modifier_Letter

Classes for Unicode categories	
`\p{Lo}`	Other_Letter
`\p{N}`	Number
`\p{Nd}`	Decimal_Digit_Number
`\p{Nl}`	Letter_Number
`\p{No}`	Other_Number
`\p{M}`	Mark
`\p{Mn}`	Non_Spacing_Mark
`\p{Mc}`	Spacing_Combining_Mark
`\p{Me}`	Enclosing_Mark
`\p{P}`	Punctuation
`\p{Pd}`	Dash_Punctuation
`\p{Ps}`	Open_Punctuation
`\p{Pe}`	Close_Punctuation
`\p{Pi}`	Initial_Punctuation
`\p{Pf}`	Final_Punctuation
`\p{Pc}`	Connector_Punctuation
`\p{Po}`	Other_Punctuation
`\p{S}`	Symbol
`\p{Sm}`	Math_Symbol
`\p{Sc}`	Currency_Symbol
`\p{Sk}`	Modifier_Symbol
`\p{So}`	Other_Symbol
`\p{Z}`	Separator
`\p{Zs}`	Space_Separator
`\p{Zl}`	Line_Separator

Classes for Unicode categories	
\p{Zp}	Paragraph_Separator
\p{C}	Other
\p{Cc}	Control
\p{Cf}	Format
\p{Co}	Private_Use
\p{Cn}	Unassigned

Classes for Unicode scripts

\p{Arabic}	\p{Kannada}
\p{Armenian}	\p{Katakana}
\p{Bengali}	\p{Khmer}
\p{Bopomofo}	\p{Lao}
\p{Cherokee}	\p{Latin}
\p{Common}	\p{Malayalam}
\p{Cyrillic}	\p{Mongolian}
\p{Devanagari}	\p{Myanmar}
\p{Ethiopic}	\p{Oriya}
\p{Georgian}	\p{Sinhala}
\p{Greek}	\p{Syriac}
\p{Gujarati}	\p{Tamil}
\p{Gurmukhi}	\p{Telugu}
\p{Han}	\p{Thaana}
\p{Hangul}	\p{Thai}
\p{Hebrew}	\p{Tibetan}
\p{Hiragana}	\p{Yi}

Examples

```
regexp_match(search_string,'abc')
```

The `regexp_match` function will return TRUE if the `search_string` anywhere has the sequence `abc`. Note the difference with the like predicate where the same criteria would need to be expressed as

```
search_string like '%abc%'
```

Escape of meta characters are done using a backslash character:

```
regexp_match(search_string,'\[abc\]')
```

would be true if `search_string` anywhere contains the string `[abc]`, (including the square brackets).

By using the boundary characters `^` and `$` it is possible to specify that a search string should start with or end with some specific characters. E.g.

```
regexp_match(search_string,'^Mimer')
```

would return true if the `search_string` started with the letters `Mimer`. For this type of searches, the database will consider using an index if appropriate.

The `regexp_match` function is collation aware. Thus

```
regexp_match('AAlborg' collate danish_1,'ålborg')
```

is true while

```
regexp_match('AAlborg' collate danish_2,'ålborg')
```

is false since a collation for danish will match `AA` to `Å`, but the level 1 collation is case insensitive which the level 2 collation is not.

This far, all of the examples given, can also be expressed with the like predicate. The following examples will deal with ranges and quantifiers which can be used to specify more complex search patterns.

To search for non-printable characters the regular expression

```
'[\x{0}-\x{1B}]'
```

could be used.

To find strings beginning with `An` or `A`, regardless of case, followed by a space and one or more arbitrary characters the pattern would be

```
'^(A|an)|A|n .+'
```

The pattern

```
'[a-zA-Z]{3}.[0-9]{3}'
```

would match a string containing three occurrences of a letter between `a` and `z` or `A` and `Z`, followed by an arbitrary character and three consecutive digits.

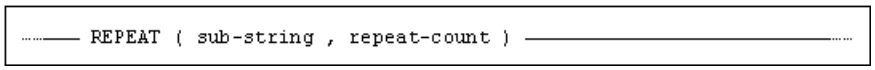
General information about the different classes for Unicode categories can be found at https://www.unicode.org/reports/tr18/ and https://www.unicode.org/reports/tr44/. Please note that these documents cover lots of functionality not supported by Mimer SQL.

REPEAT

Returns a character string composed of a specified string expression repeated a given number of times.

Syntax

Syntax for the `REPEAT` function:



`sub-string` is a character or binary string expression.

`repeat-count` is an integer expression.

Description

The result is a character or binary string consisting of `sub-string` repeated `repeat-count` times.

If the value of `repeat-count` is zero, then the result of the function is a character or binary string of length zero.

If the value of `repeat-count` is less than zero, then the result of the function is null.

If the value of either operand is null, then the result of the function is null.

Example

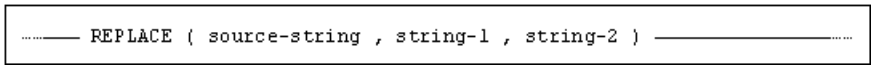
```
SET CHR_STR = REPEAT('ABC', 3);  -- sets CHR_STR to 'ABCABCABC'
```

REPLACE

Replaces all occurrences of a given string expression with another string expression in a character string.

Syntax

Syntax for the `REPLACE` function:



`source-string`, `string-1` and `string-2` are character or binary string expressions.

`source-string`, `string-1` and `string-2` must be of equal type, i.e. either all are character or all are binary.

Description

All occurrences of `string-1` found in `source-string` are replaced with `string-2`, the resulting character or binary string is returned.

If the value of any of the operands is null, then the result of the function is null.

`string-2` must not contain Unicode characters outside the Latin1 repertoire if `source-string` is of character type.

Example

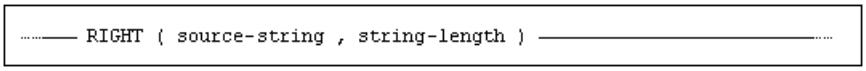
```
SET CHR_STR = REPLACE('TEST STRING', 'ST', 'NOR');  -- sets CHR_STR to
                                                    -- 'TENOR NORRING'
```

RIGHT

Returns the specified number of rightmost characters in a given character string.

Syntax

Syntax for the `RIGHT` function:



`source-string` is a character or binary string expression.

`string-length` is an integer value expressions.

Description

The rightmost `string-length` characters of `source-string` are returned.

If `count` is zero, an empty string is returned.

If `count` is less than zero, then the result of the function is null.

If the value of either operand is null, then the result of the function is null.

Example

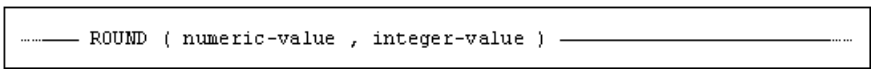
```
SET CHR_STR = RIGHT('TEST STRING', 3);  -- sets CHR_STR to 'ING'
```

ROUND

Rounds a numeric value.

Syntax

Syntax for the `ROUND` function:



`numeric-value` is an integer or a float value expression.

`integer-value` is an integer value expression.

Description

If `integer-value` is positive, the value describes the number of digits permitted in `numeric-value`, after rounding, to the right of the decimal point, if it is negative it describes the number of digits allowed to the left of the decimal point.

The value returned depends on the data type of `numeric-value`.

If the value of either operand is null, then the result of the function is null.

Returns the given numeric expression rounded to the number of places to the right of the decimal point specified by a given integer expression.

If the integer expression is negative, the numeric expression is rounded to a number of places to the left of the decimal point specified by the absolute value of the integer expression.

Examples

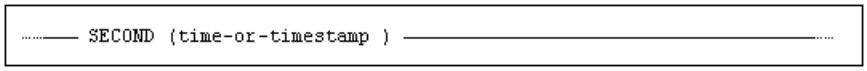
```
SET :NUM_VAL = ROUND(762.847, 2);   -- sets NUM_VAL to 762.850
SET :NUM_VAL = ROUND(762.847, 1);   -- sets NUM_VAL to 762.800
SET :NUM_VAL = ROUND(762.847, 0);   -- sets NUM_VAL to 763.000
SET :NUM_VAL = ROUND(762.847, -1);  -- sets NUM_VAL to 760.000
SET :NUM_VAL = ROUND(762.847, -2);  -- sets NUM_VAL to 800.000
SET :NUM_VAL = ROUND(7654, -2);     -- sets NUM_VAL to 7700
```

SECOND

Returns the second for the given time or timestamp expression, expressed as an integer value in the range 0-59.

Syntax

Syntax for the `SECOND` function:



`time-or-timestamp` is a time or timestamp value expression.

Description

The result is an integer value, 0 through 59, representing the second.

If the value of `time-or-timestamp` is null, then the result of the function is null.

Example

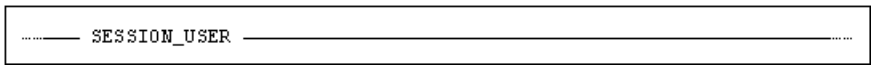
```
SET INT_VAL = SECOND(LOCALTIMESTAMP);  -- sets INT_VAL to the second number
```

SESSION_USER

Returns the name of the currently connected ident.

Syntax

Syntax for the `SESSION_USER` function:



Description

The result is the name of the current ident (i.e. the ident who established the current database connection).

The data type of the returned value is nchar varying with a maximum length of 128, with the collation SQL_IDENTIFIER.

Example

The following example returns the Program ident if entered, otherwise the session ident:

```
SET CHR_STR = COALESCE(CURRENT_PROGRAM(), SESSION_USER);
```

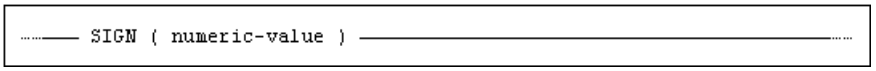
SIGN

Returns an indicator of the sign of the given numeric expression.

If the numeric expression is less than zero, -1 is returned. If the numeric expression is equal to zero, 0 is returned. If the numeric expression is greater than zero, 1 is returned.

Syntax

Syntax for the `SIGN` function:



`numeric-value` is an integer or a float value expression.

Description

The function returns an indicator of the sign of `numeric-value`. If `numeric-value` is less than zero, -1 is returned. If `numeric-value` equals zero, 0 is returned. If `numeric-value` is greater than zero, 1 is returned.

If the value of `numeric-value` is null, then the result of the function is null.

Examples

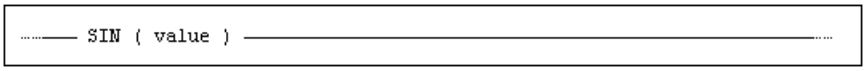
```
SET INT_VAL = SIGN(-12);  -- sets INT_VAL to -1
SET INT_VAL = SIGN(0);    -- sets INT_VAL to 0
SET INT_VAL = SIGN(12);   -- sets INT_VAL to 1
```

SIN

Returns the sine for a numeric expression.

Syntax

Syntax for the `SIN` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

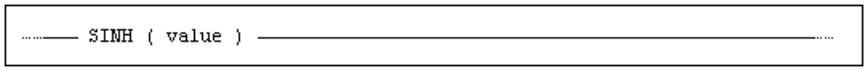
- The function returns the sine for the value expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

SINH

Returns the hyperbolic sine for a numeric expression.

Syntax

Syntax for the `SINH` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

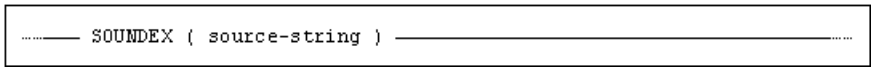
- The function returns the hyperbolic sine for the value expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

SOUNDEX

Returns a character string value containing six digits that represent an encoding of the sound of the given string expression.

Syntax

Syntax for the `SOUNDEX` function:



`source-string` is a character string expression.

Description

The function returns a character string value containing six digits that represent an encoding of the sound of `source-string`.

If `source-string` contains two or more words, they are effectively concatenated into a single word by ignoring the separating space characters.

If the `SOUNDEX` values for two strings compare to be equal then they sound the same.

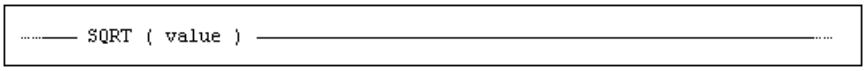
If the value of `source-string` is null, then the result of the function is null.

SQRT

Returns the square root of a numeric expression.

Syntax

Syntax for the `SQRT` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

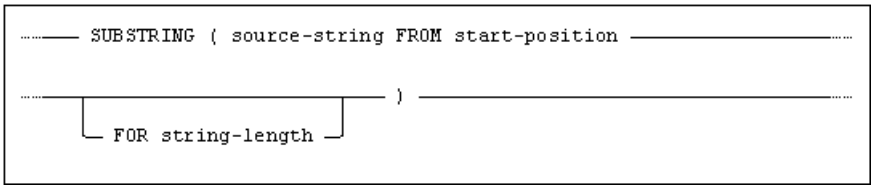
- The function returns the square root of the value. The data type for the result is double precision. Valid input values are greater than or equal to 0.
- If the value of `value` is NULL, then the result of the function is NULL.

SUBSTRING

Extracts a substring from a given string, according to specified start position and length of the substring.

Syntax

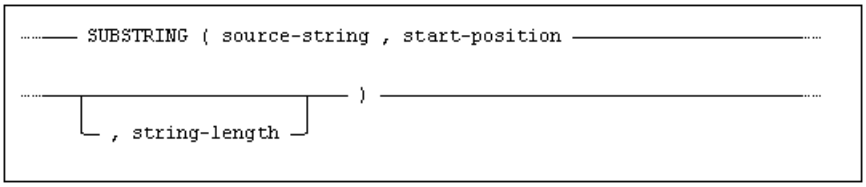
Syntax for the `SUBSTRING` function:



`source-string` is a character or binary string expression.

`start-position` and `string-length` are integer value expressions.

Alternative, comma separated syntax



Description

`SUBSTRING` returns a character or binary string containing `string-length` characters of `source-string`, starting at the character specified by `start-position`, and in the same sequence as they appear in `source-string`.

If any of these positions are before the start or after the end of `source-string`, then no character is returned for that position. If all positions are outside the source string, an empty string is returned.

The first character in `source-string` has position 1.

If the data type of source-string is variable-length character, then the result of the SUBSTRING function is a variable-length character with maximum string length equal to the maximum length of source-string. If the data type of source-string is fixed-length character, then the result of the SUBSTRING function is a variable-length character with maximum string length equal to the fixed length of source-string.

If the data type of source-string is variable-length binary, then the result of the SUBSTRING function is a variable-length binary with maximum string length equal to the maximum length of source-string. If the data type of source-string is fixed-length binary, then the result of the SUBSTRING function is a variable-length binary with maximum string length equal to the fixed length of source-string.

If string-length is negative, or if start-position is greater than the number of characters in source-string, the function fails and an error is returned.

If string-length is omitted then it is assumed to be:

```
CHAR_LENGTH(source-string) + 1 - start-position
```

i.e. the remainder of source-string, starting at start-position, is returned.

If the value of any operand is null, then the result of the function is null.

Character strings returned from a SUBSTRING function, inherit the collation from the source string.

Example

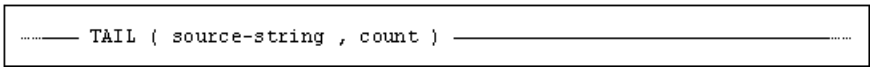
```
SET CHR_STR = SUBSTRING('Whatever' FROM 3 FOR 3); -- sets CHR_STR to 'ate'
```

TAIL

Returns the specified number of rightmost characters in a given character string.

Syntax

Syntax for the TAIL function:



source-string is a character or binary string expression.

count is an integer value expression.

Description

The rightmost count characters of source-string are returned.

If count is zero, an empty string is returned.

If count is less than zero, then the result of the function is null.

If the value of either operand is null, then the result of the function is null.

Example

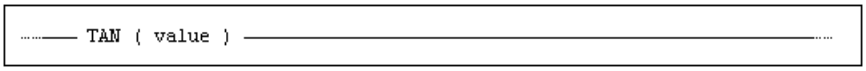
```
SET CHR_STR = TAIL('TEST STRING', 3);  -- sets CHR_STR to 'ING'
```

TAN

Returns the tangent for a numeric expression.

Syntax

Syntax for the `TAN` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

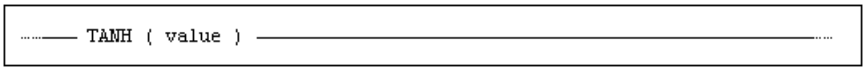
- The function returns the tangent for the value expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

TANH

Returns the hyperbolic tangent for a numeric expression.

Syntax

Syntax for the `TANH` function:



`value` is a numeric value expression. The function handles values that are within the range of a double precision expression.

Rules

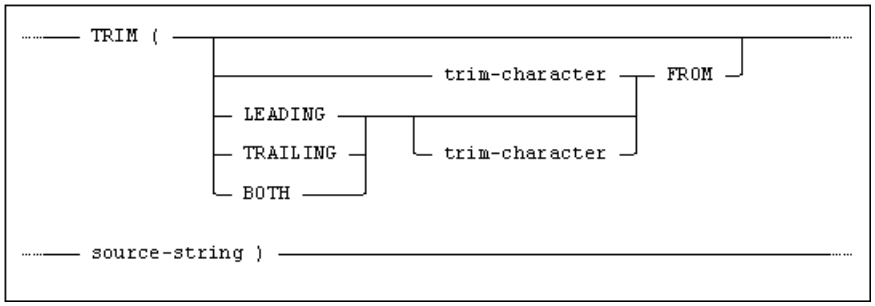
- The function returns the hyperbolic tangent for the value expressed as radians. The data type for the result is double precision.
- If the value of `value` is NULL, then the result of the function is NULL.

TRIM

Removes leading and/or trailing instances of a specified character from a string.

Syntax

Syntax for the TRIM function:



trim-character is a character or binary string expression of length 1.

source-string is a character or binary string expression.

source-string and trim-character must be of equal type, i.e. either must both be character or both binary.

Note: LEADING, TRAILING or BOTH is referred to as the trim-specification below.

Description

If trim-character is not specified, ' ' (space) is implicit for character data, and x'00' is implicit for binary data.

If trim-specification is not specified, BOTH is implicit.

If the data type of source-string is variable-length character, then the result of the TRIM function is a variable-length character with maximum string length equal to the maximum length of source-string. If the data type of source-string is fixed-length character, then the result of the TRIM function is a variable-length character with maximum string length equal to the length of source-string.

If the data type of source-string is variable-length binary, then the result of the TRIM function is a variable-length binary with maximum string length equal to the maximum length of source-string. If the data type of source-string is fixed-length binary, then the result of the TRIM function is a variable-length binary with maximum string length equal to the length of source-string.

If the length of trim-character is not 1, an error is returned.

If the value of either operand is null, then the result of the function is null.

Character strings returned from a TRIM function, inherit the collation from the source string.

Examples

```
SET CHR_STR = TRIM(' TEST ');                 -- sets CHR_STR to 'TEST'
SET CHR_STR = TRIM('T' FROM 'TEST');          -- sets CHR_STR to 'ES'
SET CHR_STR = TRIM(LEADING 'T' FROM 'TEST');  -- sets CHR_STR to 'EST'
SET CHR_STR = TRIM(TRAILING 'T' FROM 'TEST'); -- sets CHR_STR to 'TES'
```

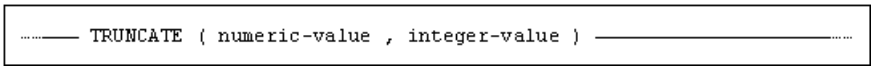
TRUNCATE

Returns the given numeric expression truncated to a number of places to the right of the decimal point specified by a given integer expression.

If the integer expression is negative, the numeric expression is truncated to a number of places to the left of the decimal point specified by the absolute value of the integer expression.

Syntax

Syntax for the `TRUNCATE` function:



`numeric-value` is an integer or a float value expression.

`integer-value` is an integer value expression.

Description

If `integer-value` is positive, the value describes the number of digits permitted in `numeric-value`, after truncation, to the right of the decimal point.

If it is negative, it describes the number of digits allowed to the left of the decimal point.

The value returned depends on the data type of `numeric-value`.

If the value of either operand is null, then the result of the function is null.

Examples

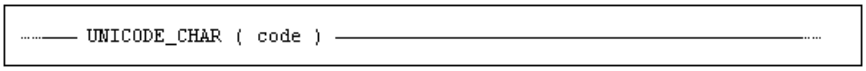
```
SET NUM_VAL = TRUNCATE(25.89, 1);   -- sets NUM_VAL to 25.80
SET NUM_VAL = TRUNCATE(25.89, -1);  -- sets NUM_VAL to 20.00
```

UNICODE_CHAR

Returns the character that has the given Unicode scalar value.

Syntax

Syntax for the `UNICODE_CHAR` function:



`code` is a numeric expression representing a Unicode scalar value.

Description

If the value of `code` represents a valid Unicode character, the function returns a single national character value, i.e. `NCHAR(1)`, otherwise an error is raised.

If the value of `code` is null, then the result of the function is null.

Example

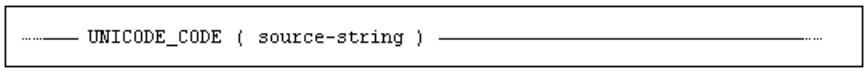
```
SET NCHR_VAL = UNICODE_CHAR(65);  -- sets NCHR_VAL to 'A'
```

UNICODE_CODE

Returns the Unicode scalar value of the leftmost character in the given string expression, as an integer.

Syntax

Syntax for the UNICODE_CODE function:



source-string is a character or binary string expression.

Description

A single INTEGER value is returned, representing a Unicode scalar value.

If the source-string contains more than one character, the Unicode scalar value of the left-most character is returned.

If the length of source-string is zero, then the result of the function is null.

If the value of source-string is null, then the result of the function is null.

Example

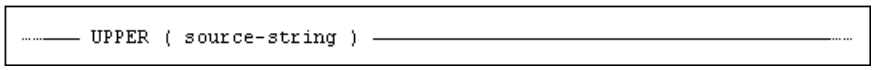
```
SET INT_VAL = UNICODE_CODE(n'A');  -- sets INT_VAL to 65
```

UPPER

Converts all lowercase letters in a character string to uppercase.

Syntax

Syntax for the UPPER function:



source-string is a character string expression.

Description

The data type of the result is the same as the data type of source-string.

source-string is either in character or national character (i.e. Unicode) format.

If the value of source-string is null, then the result of the function is null.

Note: The length of a result may be longer or shorter than the input value. This means that using UPPER on a column may cause data truncation.

USER

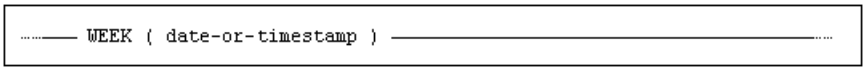
Returns the same value as CURRENT_USER. We recommend that you use CURRENT_USER, see *CURRENT_USER* on page *99*.

WEEK

Returns the week of the year for the given date or timestamp expression, expressed as an integer value in the range 1-53.

Syntax

Syntax for the `WEEK` function:



`date-or-timestamp` is a date or timestamp value expression.

Description

The result is an integer value, 1 through 53, representing the week number in the year, calculated in accordance with the ISO 8601 standard. (The year's first week with 4 or more days is week 1.)

If the value of `date-or-timestamp` is null, then the result of the function is null.

Example

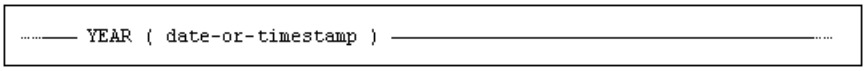
```
SET INT_VAL = WEEK(CURRENT_DATE);  -- sets INT_VAL to the week number
                                   -- of the current year
```

YEAR

Returns the year for the given date or timestamp expression, expressed as an integer value in the range 1-9999.

Syntax

Syntax for the `YEAR` function:



`date-or-timestamp` is a date or timestamp value expression.

Description

The result is an integer value, 1 through 9999, representing the year.

If the value of `date-or-timestamp` is null, then the result of the function is null.

Example

```
SET INT_VAL = YEAR(CURRENT_DATE);  -- sets INT_VAL to the year number
                                   -- of the current year
```

Standard Compliance

This section summarizes standard compliance for scalar functions.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F052, “Intervals and datetime arithmetic”. Feature F555, “Enhanced seconds precision” LOCALTIME and LOCALTIMESTAMP functions with fractions of seconds. Feature T176, “Sequence generator support”. Feature T312, “OVERLAY function”. Feature T441, “Support for ABS and MOD functions”. Feature T621, “Enhanced numeric functions”. Feature T622, “Trigonometric functions”. Feature T624, “Common logarithm functions”.

Standard	Compliance	Comments
	Mimer SQL extension	Support for: ASCII_CHAR ASCII_CODE ATAN2 BEGINS BUILTIN.BEGINS_WORD BUILTIN.MATCH_WORD BUILTIN.UTC_TIMESTAMP COT CURRENT_PROGRAM CURRENT VALUE DAYOFMONTH DAYOFWEEK DAYOFYEAR DEGREES HOUR INDEX_CHAR IRAND LEFT LOCATE MONTH PASTE QUARTER RADIANS REGEXP_MATCH REPEAT REPLACE ROUND SECOND SIGN SOUNDEX TAIL TRUNCATE UNICODE_CHAR UNICODE_CODE WEEK YEAR is a Mimer SQL extension.

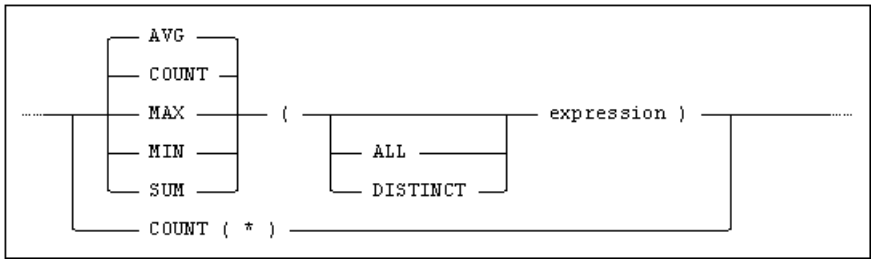
Set Functions

Set functions are pre-defined functions used in select specifications. They operate on the set of values in one column of the result of the `SELECT` statement, or on the subset in a group if the statement includes a `GROUP BY` clause.

The result of a set function is a single value for each operand set.

Syntax for Set Functions

The general syntax for a set function is:



AVG

Returns the average of the values in the set.

Note: `AVG` can only be applied to numerical and interval values.

COUNT

Returns the number of values in the set.

MAX

Returns the largest value in the set.

MIN

Returns the smallest value in the set.

SUM

Returns the sum of the values in the set.

Note: `SUM` can only be applied to numerical and interval values.

Examples

```
SELECT MIN(PRICE) AS INEXPENSIVE, MAX(PRICE) AS EXPENSIVE
FROM   ROOM_PRICES WHERE HOTELCODE = 'LAP';

SELECT HOTELCODE, AVG(PRICE) AS AVERAGE_PRICE
FROM   ROOM_PRICES
GROUP BY HOTELCODE;

SELECT COUNT(*) FROM SOME_TABLE;
```

Operational Mode

The operational mode of a set function is determined by the use of the keywords ALL and DISTINCT.

When ALL is specified or no keyword is used:

- Any duplicate values in the operand set are retained.

When DISTINCT is specified:

- Redundant duplicate values are eliminated from the operand set before the function is applied.
- The result of the set function must not be combined with other terms using binary arithmetic operators.
- For the set functions MAX and MIN, the DISTINCT keyword makes no difference to the result. (The same value will be returned with or without DISTINCT.)

Null Values

For all set functions except COUNT(*), any null values in the operand set are eliminated before the set function is applied, regardless of whether DISTINCT is specified or not.

The special form COUNT(*) returns the number of rows in the result table, including any null values. The keywords ALL and DISTINCT may not be used with this form of COUNT.

If the operand set is empty, the COUNT function returns the value zero. All other functions return null for an empty operand set.

The COUNT function returns an INTEGER. The MAX and MIN functions return a value with the same type and precision as the operand. The precision of the result returned by SUM and AVG is considered below.

Restrictions

Column references in the argument of a set function may not address view columns which are themselves derived from set functions.

The argument of a set function must contain at least one column reference and cannot contain any set function references. If the column is an outer reference, then the expression should not include any operators.

If a set function contains a column that is an outer reference, then the set function must be contained in a subquery of a HAVING clause.

Results of Set Functions

When the argument of a set function is a numerical value, the precision and scale of the set function result is evaluated in accordance with the rules given below. If the argument is an expression, the expression is first evaluated as described in *Expressions* on page *141* before the set function is applied.

Evaluating Set Functions

	FLOAT(p')	INTEGER(p')	DECIMAL(p',s')
SUM	FLOAT(p)[a]	INTEGER(p)[b]	DECIMAL(p,s)[c]
AVG	FLOAT(p)[a]	INTEGER(p)[d]	DECIMAL(p,s)[e]
MAX, MIN	FLOAT(p)[d]	INTEGER(p)[d]	DECIMAL(p,s)[f]
COUNT	INTEGER(10)	INTEGER(10)	INTEGER(10)

a. p=max(15, p')
b. p=min(45, 10+p')
c. p=min(45, 10+p') s=s'
d. p=p'
e. p=min(45, 10+p') s=p-(p'-s')
f. p=p' s=s'

The following examples show how some set functions are evaluated.

`AVG(SMALLINT)` gives `SMALLINT`

`AVG(INTEGER)` gives `INTEGER`

`AVG(DECIMAL(38,10))` gives `DECIMAL(45,17)`

`AVG(DECIMAL(4,2))` gives `DECIMAL(14,12)`

`AVG(INTERVAL YEAR(2) TO MONTH)` gives `INTERVAL YEAR(2) TO MONTH`

`SUM(SMALLINT)` gives `INTEGER(15)`

`SUM(INTEGER)` gives `INTEGER(20)`

`SUM(DECIMAL(38,10))` gives `DECIMAL(45,10)`

`SUM(DECIMAL(4,2))` gives `DECIMAL(14,2)`

`SUM(INTERVAL YEAR(2) TO MONTH)` gives `INTERVAL YEAR(2) TO MONTH`

Note: Often, the average of a series of integers is required as a decimal rather than an integer. This may be achieved by casting the value to a decimal using the `CAST` function.
For example, if the values in the integer column `COL` are `1`, `3` and `6`, then `AVG(COL)` returns `3` but `AVG(CAST(COL as decimal(14,4)))` returns `3.33333333333333`.

Alternatively, multiply the `AVG` argument by 1.0, i.e. `AVG(COL * 1.0)`.

Standard Compliance

This section summarizes standard compliance for set functions.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F441, “Extended set function support”. Feature F561, “Full value expressions” use of DISTINCT expression in set function, where expression is not a column.

Chapter 9

Expressions and Predicates

This chapter discusses general value specifications, known as expressions; and conditional statements, known as predicates.

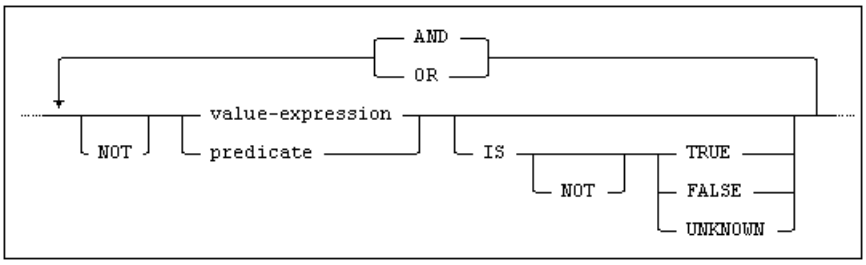
Expressions

Expressions are used in a variety of contexts within SQL statements, particularly in search condition predicates and the `SET` clause in `UPDATE` statements respectively.

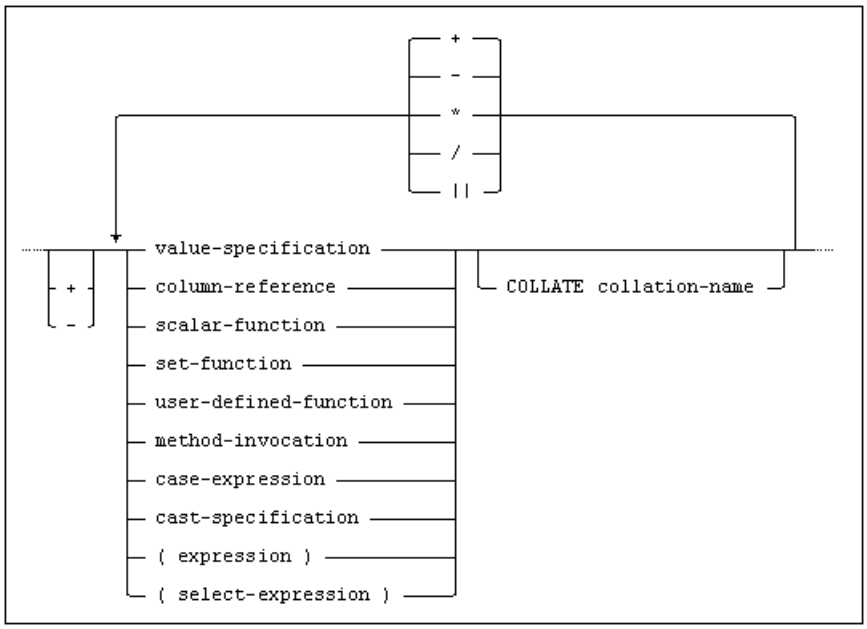
An expression always evaluates to a single value.

Syntax

The syntax of an `expression` is:



where a `value-expression` is as follows:



Note: A `user-defined-function` is created by using the `CREATE FUNCTION` statement.

Note: In this position, the `COLLATE` clause's purpose is to specify the result's collation. E.g. `MIN(col_swe) collate english_1` will evaluate `MIN` according to `col_swe`'s collation, then the result will have an english_1 collation attribute.

Unary Operators

A unary operator operates on only one operand.

The prefix operator + (unary plus) does not change its operand.

The prefix operator – (unary minus) reverses the sign of its operand.

Binary Operators

A binary operator operates on two operands.

The binary operators specify addition (+), subtraction (–), multiplication (*) and division (/) for numerical operands, and concatenation (||) for string operands.

Operands

When a column name is used as an operand, it represents the single value contained in the column for the row currently addressed when the expression is evaluated.

The column name may be qualified by the name of the table or view, see *Identifiers* on page *38*.

Evaluating Arithmetical Expressions

Expressions within parentheses are evaluated first. When the order of evaluation is not specified by parentheses, the customary arithmetical rules apply, i.e. multiplication and division are performed before addition and subtraction and operators with the same precedence are applied from left to right.

If any operand in an expression is null, the whole expression evaluates to null. No other expressions evaluate to null. Division by zero results in a run-time error.

Arithmetical expressions with mixed numerical and character data are illegal.

Note: Where host variables are used in expressions, type conversion may result in apparently incompatible data types being accepted, see *Data Types in SQL Statements* on page *44*.

The type and precision of the result of an arithmetical expression is determined in accordance with the rules described below. If there are more than two operands in an expression, the type and precision of the result is derived in accordance with the sequence in which the component binary operations are performed.

Result Data Types

In descriptive terms, the rules are as follows:

- **If any of the operands is a DOUBLE PRECISION, the result is DOUBLE PRECISION.**

  For all arithmetic expressions, the result is always a `DOUBLE PRECISION`, no matter what other data types are involved.

- **Otherwise, if any of the operands is a REAL, the result is REAL.**

  For all arithmetic expressions, the result is always a `REAL`, except for if the other operand is `DOUBLE PRECISION`, then the result is `DOUBLE PRECISION`.

- **Otherwise, if any of the operands is FLOAT(p), the result is FLOAT(p).**

  For all arithmetic expressions, the precision of the result is the highest operand precision. However, the precision of a `FLOAT(p)` result is never less than 16.

- **Otherwise, if all the operands are integer, the result is integer.**

  The integer data types `SMALLINT`, `INTEGER` and `BIGINT` are used whenever possible (when the result fits).

  For addition, subtraction and multiplication with both operands of type `SMALLINT`, `INTEGER` or `BIGINT`, the result data type is the data type of the highest operand, except for operations on two `SMALLINT`'s, where the result is an `INTEGER`. For division of `SMALLINT`, `INTEGER` or `BIGINT`, the result data type is the data type of the dividend.

  For addition, subtraction and multiplication with at least one operand that is `INTEGER(p)`, where the precision is less than 19, the result data type is an `INTEGER` or `BIGINT`, the highest operand decides.

  For addition, subtraction and multiplication with at least one operand that is `INTEGER(p)` where the precision is greater than 18, the result data type is an `INTEGER(p)`. For addition and subtraction, the precision of the result is the precision of the highest operand + 1. For multiplication, the precision of the result is the sum of the precision of the operands. However, the precision may never exceed 45 for the result of any operation.

For division, the result data type is mainly the data type of the dividend. If the dividend is an `INTEGER(p < 19)`, the result data type is a `SMALLINT` when p < 5, `INTEGER` when p < 10 or `BIGINT` when p < 19. If the dividend is an `INTEGER(p > 18)`, the result data type is an `INTEGER(p)`, where the precision is the same as of the dividend.

- **Otherwise, if all the operands are decimal, or decimal and integer operands are mixed, the result is decimal.**

  For expressions mixing decimal and integer operands, the integer operand is treated like a decimal with scale set to 0. `INTEGER(p)` is treated as `DECIMAL(p,0)`, `SMALLINT` is treated as `DECIMAL(5,0)`, `INTEGER` is treated as `DECIMAL(10,0)` and `BIGINT` is treated as `DECIMAL(19,0)`.

  For addition and subtraction, the number of positions to the left of the decimal point (i.e. the difference between precision and scale) in the result is the greatest number of positions in any operand plus 1. The scale of the result is the greatest scale of any of the operands. The precision may not exceed 45.

  For multiplication, the precision of the result is the sum of the precisions of the operands.

  The scale of the result is the sum of the scales of the operands. Neither the precision nor the scale may exceed 45. If the value of the result does not fit into the precision and scale, overflow occurs.

  For division, the precision of the result is the sum of the precisions of the operands. The precision is however never less than 15 and may not exceed 45.

  The scale of the result is calculated as the precision of the result, minus the number of positions to the left of the decimal point in the dividend, minus the scale of the divisor.

Evaluating String Expressions

The result of a string concatenation expression is a string containing the first operand string directly followed by the second.

The following rules apply:

- If string literals or fixed-length host variables are concatenated, any trailing blanks in the operands are retained.
- If a fixed-length character column value is directly concatenated with another string, any trailing blanks in the column value up to the defined fixed length of the column are retained.
- If a variable-length character column value is directly concatenated with another string, any trailing blanks in the column value up to the actual length of the column value are retained.
- If two character values are concatenated, the result will be a variable-length character value.
- If a character value and a national character value are concatenated, the result will be a variable-length national character value.
- If either of the operands in a concatenation expression is null, the result of the expression is null.

- When concatenating string expressions, the resulting string's collation depends on whether and where a collation has been specified:
  - If no collation(s) have been specified for the column-definition, in a domain or explicitly in the concatenation statement, then the resulting string has the Mimer SQL default collation. See *Appendix B Character Sets*.
  - If one string has a specific collation and the other(s) do not then they are coerced into having the specific collation.
  - If the strings have specific but differing collations, an error will be raised.

  For more information, see the *Mimer SQL User's Manual, Chapter 4, Collations*.

Select Specification

A select specification can be used as an expression. This is commonly known as scalar subqueries. A scalar subquery may not return more than one value. The result of an empty subquery is null.

Examples

```
SET total = (SELECT COUNT(*) FROM categories)

SELECT c.surname, c.forename,
       (SELECT COUNT(*) FROM orders
        WHERE customer_id = c.customer_id) AS orders
FROM customers AS c
```

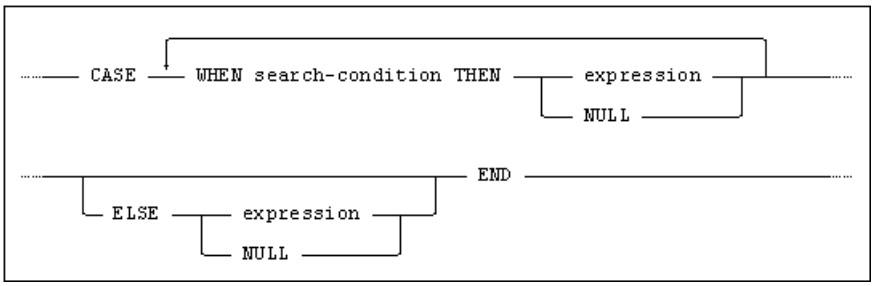
The last example shows a correlated subquery i.e. a subquery with a reference to a column in a table not present in the subquery itself.

CASE Expression

With a `CASE` expression, it is possible to specify a conditional value. Depending on the result of one or more conditional expressions, the `CASE` expression can return different values.

A `CASE` expression can be in one of the following two forms.

CASE Expression First Form



Rules

The following rules apply to `CASE` expressions:

- If one or more `search-conditions` are true, then the result of the `CASE` expression is the result of the first (left-most) `WHEN` clause which has a `search-condition` that is true.

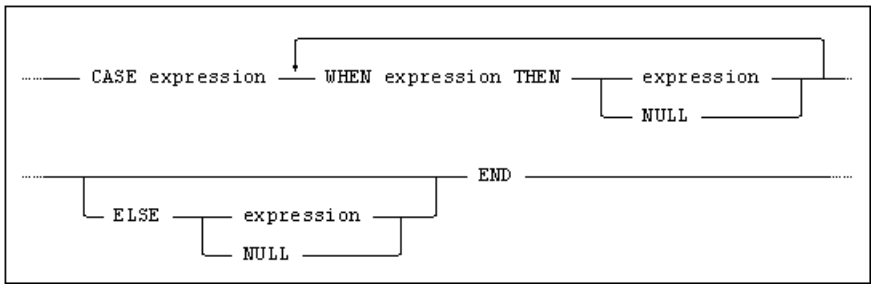
- If none of the search-conditions are true, then the result of the CASE expression is the result of the explicit or implicit ELSE clause.
- If no ELSE clause is specified then ELSE NULL is implicit.
- At least one result in a CASE expression must express a value different from null.

See *Result Data Types* on page *84* for a description of how the data type of the result of the CASE expression is determined.

Example

```
CASE WHEN col1 < 10  THEN 1
     WHEN col1 < 100 THEN 2
     ELSE 3
END
```

CASE Expression Second Form



Rules

The following rules apply to CASE expressions:

- If no ELSE clause is specified then ELSE NULL is implicit.
- A case expression using the second form will do an equality comparison between the expression preceding the first when clause and each expression in the when clauses, going from left to right, until one comparison evaluates to true in which case the expression in the THEN part of the when clause is returned. If no comparison evaluates to true, the expression in the else clause is returned.
- At least one result in a CASE expression must express a value different from null.
- The expression proceeding the first WHEN clause and all expressions in the WHEN clauses must be comparable.
- All expressions in the THEN and ELSE clauses must be comparable or be NULL.

See *Result Data Types* on page *84* for a description of how the data type of the result of the CASE expression is determined.

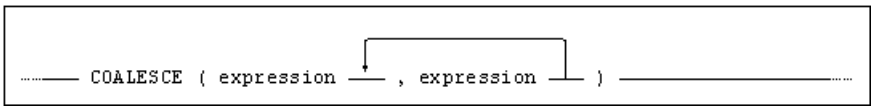
Example

```
CASE col1 WHEN 0 THEN NULL
          WHEN -1 THEN -999
          ELSE col1
END
```

Short Forms for CASE

There are two short forms for special `CASE` expressions: `COALESCE` and `NULLIF`.

COALESCE



where:

```
COALESCE(x1,x2)
```

is equivalent to:

```
CASE WHEN x1 IS NOT NULL THEN x1 ELSE x2
END
```

and:

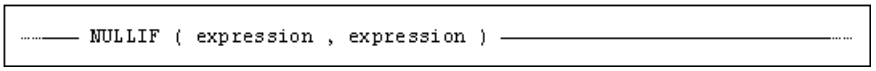
```
COALESCE(x1,x2,...,xn)
```

is equivalent to:

```
CASE WHEN x1 IS NOT NULL THEN x1
ELSE COALESCE(x2,...,xn) END
```

I.e. the `COALESCE` expression returns the value of the first non-null operand, found by working from left to right, or null if all the operands equal null.

NULLIF



where

```
NULLIF(x1, x2)
```

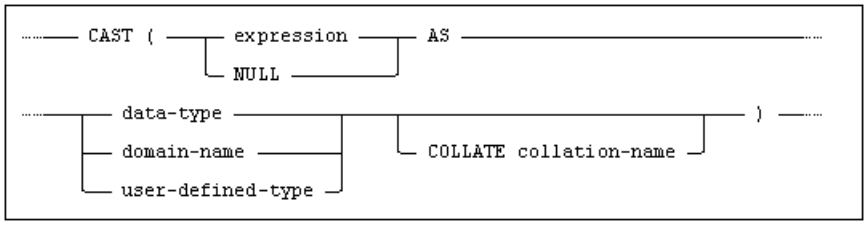
is equivalent to

```
CASE WHEN x1 = x2 THEN NULL ELSE x1 END
```

I.e. if the operands are equal, the `NULLIF` expression has the value null, otherwise it has the value of the first operand.

CAST Specification

With the CAST specification it is possible to specify a data type conversion. CAST converts the value of an expression to a specified data type.



Rules

The following rules apply to CAST:

- data-type can be any (cast compatible) SQL data type supported by Mimer SQL.

  The table below lists cast compatibility.

Source data type	Target data type
INTEGER SMALLINT BIGINT	INTEGER, BIGINT, SMALLINT INTEGER(p) DECIMAL/NUMERIC REAL, DOUBLE PRECISION/FLOAT FLOAT(p) CHARACTER, VARCHAR NCHAR, NVARCHAR BINARY, VARBINARY INTERVAL YEAR-MONTH INTERVAL DAY-TIME
INTEGER(p) DECIMAL/NUMERIC	INTEGER, BIGINT, SMALLINT INTEGER(p) DECIMAL/NUMERIC REAL, DOUBLE PRECISION/FLOAT FLOAT(p) CHARACTER, VARCHAR NCHAR, NVARCHAR INTERVAL YEAR-MONTH INTERVAL DAY-TIME
REAL DOUBLE PRECISION/ FLOAT FLOAT(p)	INTEGER, BIGINT, SMALLINT INTEGER(p) DECIMAL/NUMERIC REAL, DOUBLE PRECISION/FLOAT FLOAT(p) CHARACTER, VARCHAR NCHAR, NVARCHAR

CHARACTER VARCHAR	INTEGER, BIGINT, SMALLINT INTEGER(p) DECIMAL/NUMERIC REAL, DOUBLE PRECISION/FLOAT FLOAT(p) CHARACTER, VARCHAR NCHAR, NVARCHAR BOOLEAN BINARY, VARBINARY DATE TIME TIMESTAMP INTERVAL YEAR-MONTH INTERVAL DAY-TIME
CLOB	CLOB
NCHAR NVARCHAR	INTEGER, BIGINT, SMALLINT INTEGER(p) DECIMAL/NUMERIC REAL, DOUBLE PRECISION/FLOAT FLOAT(p) CHARACTER, VARCHAR NCHAR, NVARCHAR BOOLEAN DATE TIME TIMESTAMP INTERVAL YEAR-MONTH INTERVAL DAY-TIME
NCLOB	NCLOB
DATE	DATE TIMESTAMP CHARACTER, VARCHAR NCHAR, NVARCHAR
TIME	TIME TIMESTAMP CHARACTER, VARCHAR NCHAR, NVARCHAR
TIMESTAMP	DATE TIME TIMESTAMP CHARACTER, VARCHAR NCHAR, NVARCHAR
INTERVAL YEAR-MONTH	INTERVAL YEAR-MONTH INTEGER, BIGINT, SMALLINT INTEGER(p) DECIMAL/NUMERIC CHARACTER, VARCHAR NCHAR, NVARCHAR

INTERVAL DAY-TIME	INTERVAL DAY-TIME INTEGER, BIGINT, SMALLINT INTEGER(p) DECIMAL/NUMERIC CHARACTER, VARCHAR NCHAR, NVARCHAR
BOOLEAN	BOOLEAN CHARACTER, VARCHAR NCHAR, NVARCHAR
BINARY VARBINARY	BINARY, VARBINARY INTEGER, BIGINT, SMALLINT
BLOB	BLOB

- When converting a value to fixed-length character, the value of the source expression is padded with trailing spaces, if the length of the converted value is shorter than the length of the target data type.
- When converting a value to variable-length character, no trailing spaces are padded.
- When converting a value to fixed-length binary, the value of the source expression is padded with trailing 0's, if the length of the converted value is shorter than the length of the target data type.
- When converting a value to variable-length binary, no trailing 0's are padded.
- A character value can be converted to a national character value.
- A national character value can be converted to a character value. If the national character value contains non-Latin1 characters, an error is raised.
- Character values can be converted to a numeric data type if the character string consists of a valid literal representation of the target data type.
- Character values can be converted to a DATETIME or INTERVAL data type if the character string consists of a valid literal representation of the target data type.
- When a DATE value is converted to a TIMESTAMP, the HOUR, MINUTE and SECOND fields of the target are set to zero. The other fields are set to the corresponding values in the source expression.
- When a TIME value is converted to a TIMESTAMP, the respective values for the YEAR, MONTH and DAY fields of the target are obtained by evaluating CURRENT_DATE. The other fields are set to the corresponding values in the source expression.
- When a TIMESTAMP value is converted to a DATE or TIME, the fields of the target are set to the corresponding values in the source expression. Any values in the source expression for which there are no corresponding fields in the target are ignored.
- When converting from a INTERVAL to an exact numeric value, it must be possible to represent the INTERVAL value as an exact numeric value without the loss of leading significant digits.
- When converting from an exact numeric value to an INTERVAL, it must be possible to represent the exact numeric as an INTERVAL value without the loss of leading significant digits.

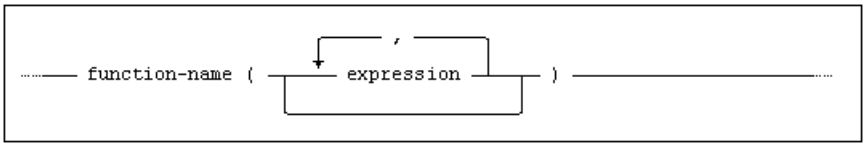
- If `CAST` is applied on `NULL`, or if `expression` results in null, then `CAST` returns null.
- Character values can be converted to a `BOOLEAN` data type provided expression contains the string `TRUE` or `FALSE` regardless of case.
- When converting a boolean value to character, the boolean value `TRUE` is converted to the string `TRUE` and the boolean value `FALSE` is converted to the string `FALSE`.

Example

```
SELECT CAST(floatcol AS DECIMAL(15,3)),
       CAST(charcol AS VARCHAR(10)),
       CAST(intcol AS CHAR(15)),
       CAST(decimcol AS DOUBLE PRECISION)
FROM types_tab;
```

User-Defined Function

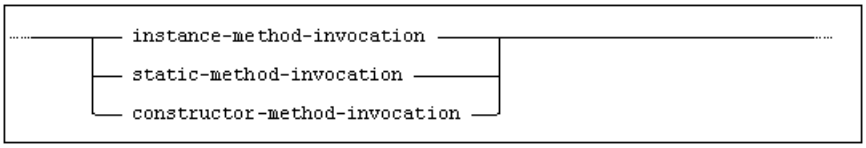
The syntax to invoke a user-defined function is:



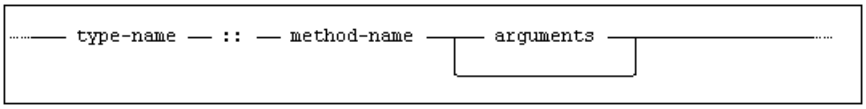
The function name may be qualified by a schema name in the normal manner.

Method Invocation

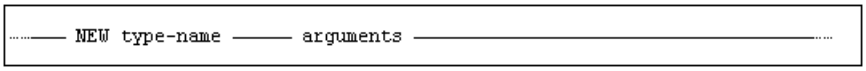
The syntax for a `method-invocation` is:



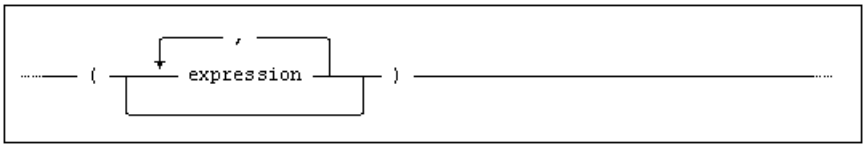
and `static-method-invocation` is:



and `constructor-method-invocation` is:



and `arguments` are:



The type name may be qualified by a schema name in the normal manner. A method name however can not be qualified by a schema name in the context of a method invocation.

An instance method invocation requires that there is an instance of the user-defined type on which the method is defined. An instance is created by using the initializer function if it is distinct.

Note that if a method invocation returns a user-defined type, it is possible to use this result as an instance for further invocations.

Standard Compliance

This section summarizes standard compliance for expressions.

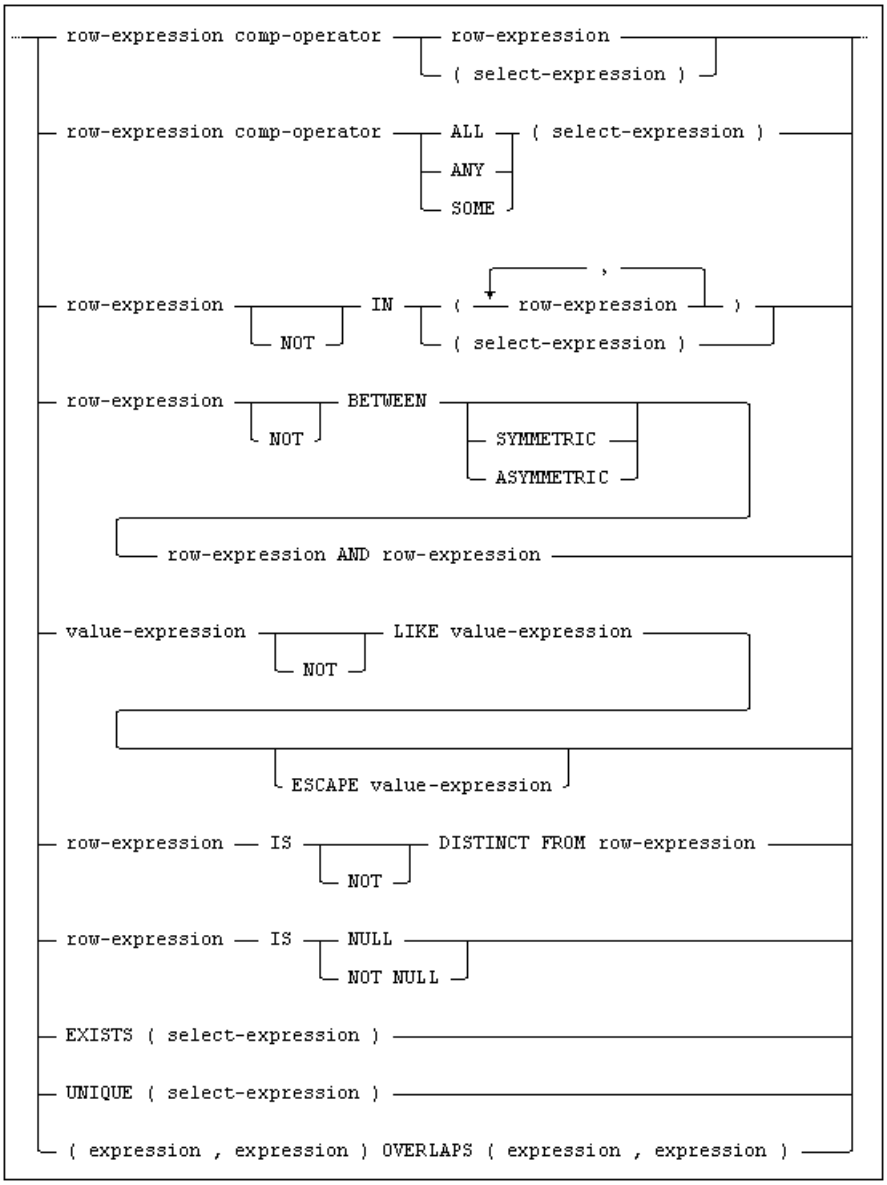
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F052, “Intervals and Datetime arithmetic”. Feature F251, “Domain support” use of domain as target specification in cast expressions. Feature F690, “Collation support” support for collate clause. Feature P002, “Computational completeness”. Feature T031, “Boolean data type”.

Predicates

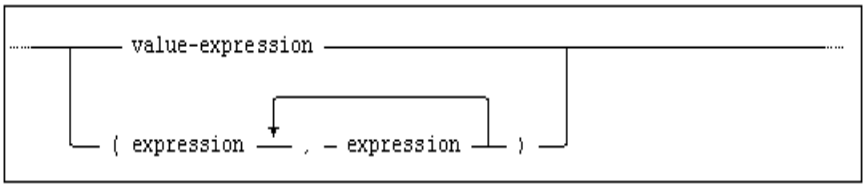
A predicate is a single conditional expression which evaluates to either true, false or unknown. Predicates are used in constructing search conditions, see *Search Conditions* on page *165*.

Predicate Syntax

The general predicate syntax is shown below:



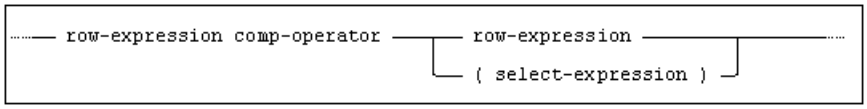
where `row-expression` is:



Each individual predicate construction is explained in more detail in the following sections.

The Basic Predicate

A basic predicate compares a value with one and only one other value, and has the syntax:



The comparison operators, `comp-operator`, are described in *Comparison Operators* on page *73*.

The expressions on either side of the comparison operator must have compatible data types, see *Comparisons* on page *80*.

Within the context of a basic predicate, a `select-expression` must result in either an empty set or a single value.

The result of the predicate is unknown if either of the expressions used evaluates to null, or if the `select-expression` used results in an empty result set.

A comparison involving row expressions requires that the two row expressions have the same number of elements, and that each element in the first row expression is comparable with the corresponding element in the second row expression.

The comparison will be done from left to right and will continue until all elements have been compared or the predicate is false.

As an example, consider this predicate

```
(a1,a2,a3) < (b1,b2,b3)
```

which is equivalent to

```
a1 < b1 or (a1 <= b1 and a2 < b2) or (a1 <= b1 and a2 <= b2 and a3 < b3)
```

For instance

```
(1,date '1956-04-23', false) < (1,date '1956-04-23', true)
```

would evaluate to TRUE since FALSE is less than TRUE.

Null values are handled analogously with comparisons with single values. Thus

```
(1,cast(null as int)) < (1,2)
```

would become null, but

```
(1,cast(null as int)) < (2,2)
```

would become true since the second elements are never compared in this case.

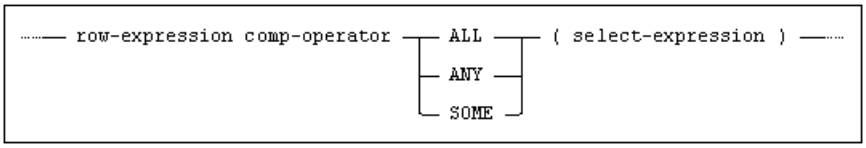
Row-expression examples

```
select * from tabA where (c1, c2) = (select k1, k2 from tabB fetch 1);

select * from tabA where (abs(c1), c2) > (c3, lower(c4));
```

The Quantified Predicate

A quantified predicate compares an expression with a set of values addressed by a subquery (as opposed to a basic predicate which compares two single-valued expressions).

The form of the quantified expression is:



The comparison operators, `comp-operator`, are described in *Comparison Operators* on page *73*.

Within the context of a quantified predicate, a `select-expression` must result in either an empty set or a set of single values.

ALL

The result is true if the select-specification results in an empty set or if the comparison is true for every value returned by the `select-expression`.

The result is false if the comparison is false for at least one value returned by the `select-expression`.

The result is unknown if any of the values returned by the `select-expression` is null and no value is false.

ANY or SOME

The keywords `ANY` and `SOME` are equivalent.

The result is true if the comparison is true for at least one value returned by the `select-expression`.

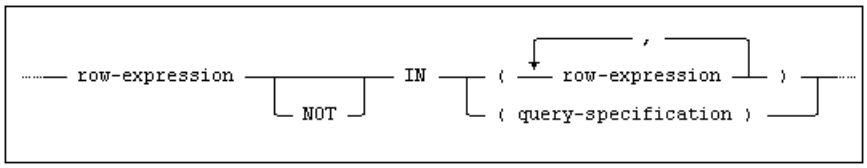
The result is false if the select results in an empty set or if the comparison is false for every value returned by the `select-expression`.

The result is unknown if any of the values returned by the select is null and no value is true.

Quantified predicates may always be replaced by alternative formulations using `EXISTS`, which can often clarify the meaning of the predicates.

The IN Predicate

The `IN` predicate tests whether a value is contained in a set of discrete values and has the form:



If the set of values on the right hand side of the comparison is given as an explicit list, an `IN` predicate may always be expressed in terms of a series of basic predicates linked by one of the logical operators `AND` or `OR`:

IN predicate	Equivalent basic predicates
`x IN (a,b,c)`	`x = a OR x = b OR x = c`
`x NOT IN (a,b,c)`	`x <> a AND x <> b AND x <> c`

If the set of values is given as a `select-expression`, an `IN` predicate is equivalent to a quantified predicate:

IN predicate	Equivalent quantified predicates
`x IN (subquery)`	`x = ANY (subquery)`
`x NOT IN (subquery)`	`x <> ALL (subquery)`

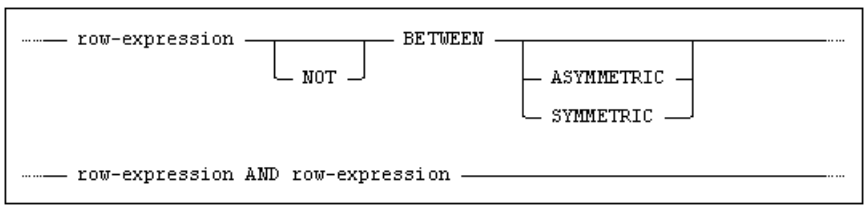
The result of the `IN` predicate is unknown if the equivalent predicates give an unknown result.

Note: `NOT IN` is undefined if the subquery result contains a null value. E.g. `SELECT * FROM tab WHERE 3 NOT IN (2, <null>, 4)` will return an empty result set.

The BETWEEN Predicate

A `BETWEEN` predicate tests whether or not a value is within a range of values (including the given limits).

It has the form:



The `BETWEEN` predicate can always be expressed in terms of basic predicates.

If neither `SYMMETRIC` nor `ASYMMETRIC` is specified, then `ASYMMETRIC` is implicit.

Thus:

Between predicate	Equivalent basic predicates
`x BETWEEN a AND b`	`x >= a AND x <= b`
`x NOT BETWEEN a AND b`	`x < a OR x > b`
`x BETWEEN SYMMETRIC a AND b`	`(x >= a AND x <= b) OR (x >= b AND x <= a)`
`x NOT BETWEEN SYMMETRIC a AND b`	`(x > a AND x > b) OR (x < a AND x < b)`

Examples

Expression	Result
`2 BETWEEN 1 AND 3`	`TRUE`
`2 BETWEEN 3 AND 1`	`FALSE`
`2 BETWEEN SYMMETRIC 1 AND 3`	`TRUE`
`2 BETWEEN SYMMETRIC 3 AND 1`	`TRUE`
`(1,2) BETWEEN (1,1) AND (1,3)`	`TRUE`
`(1,2) BETWEEN (1,1) AND (1,0)`	`FALSE`

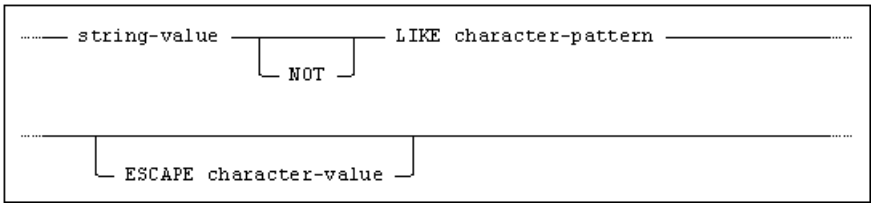
All expressions in the predicate must have compatible data types.

The result of the predicate is unknown if the equivalent basic predicates give an unknown result.

The LIKE Predicate

The `LIKE` predicate compares the value in a string expression with a character string pattern which may contain wildcard characters (meta-characters).

It has the form:



The `string-value` on the left hand side of the `LIKE` operator must be a string expression.

The `character-pattern` on the right hand side of the `LIKE` operator is a string expression.

The escape `character-value` must be a string expression of length 1. To search for the escape character itself it must appear twice in immediate succession in the like pattern.

Meta-characters/Wildcards

The following meta-characters (wildcards) may be used in the `character-pattern`:

`_` stands for any single character

`%` stands for any sequence of zero or more characters.

Note: Wildcard characters are only used as such in LIKE predicates. In any other context, the characters _ and % have their exact values.

Escape Characters

The optional escape character is used to allow matching of the special characters _ and %. When the escape character prefixes _ and %, they are interpreted without any special meaning.

An escape character used in a pattern string may only be followed by another escape character or one of the wildcard characters, unless it is itself escaped (i.e. preceded by an escape character).

Examples

LIKE predicate	Matches
`LIKE 'A%'`	any string beginning with A
`LIKE '_A%'`	any string, where the second character is A
`LIKE '%A%'`	any string containing an A
`LIKE '%\%%' ESCAPE '\'`	any string containing a %
`LIKE '%A\%\\' ESCAPE '\'`	any string ending with A%\
`LIKE '_ABC'`	any 4-character string ending in ABC

A LIKE predicate where the pattern string does not contain any wildcard characters is essentially equivalent to a basic predicate using the = operator.

The comparison strings in the LIKE predicate are not conceptually padded with blanks, in contrast to the basic comparison.

Thus:

`'artist  ' = 'artist'` is true

`'artist  ' LIKE 'artist  '` is true

`'artist  ' LIKE 'artist%'` is true

but

`'artist  ' LIKE 'artist'` is false

Begins With

LIKE predicates, addressing the “begins with” functionality, are very common.

However, when a parameter marker is used for the LIKE pattern, the SQL compiler can not determine the LIKE pattern characteristics, and possible optimizations will not be applied. The built-in function BEGINS will overcome this issue. See *BEGINS* on page *93* for information.

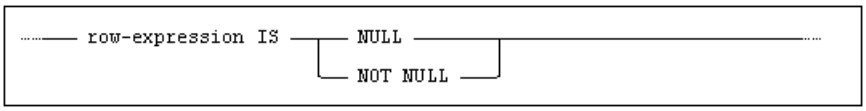
Regular Expressions

Compared to LIKE, the regular expression provides a much more flexible way to match strings of text, such as complex patterns of characters.

Use the REGEXP_MATCH function to do regular expression searches. See *REGEXP_MATCH* on page *117* for information.

The NULL Predicate

The NULL predicate is used to test if the specified expression is the null value, and has the form:



The result of the NULL predicate is never unknown.

Evaluation rules for the NULL predicate:

x	x IS NULL	x IS NOT NULL	NOT x IS NULL	NOT x IS NOT NULL
null	True	False	False	True
not null	False	True	True	False
(null, null)	True	False	False	True
(not null, null)	False	False	True	True
(not null, not null)	False	True	True	False

The use of composite expressions in NULL predicates provides a shorthand for testing whether any of the operands is null.

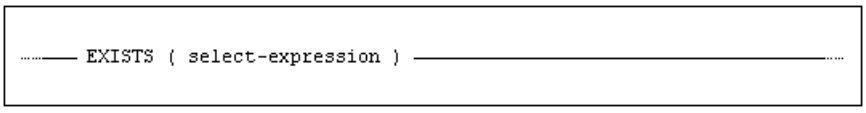
Thus the predicate A=B IS NULL is an alternative to A IS NULL OR B IS NULL.

Note: The actual operator(s) used in expressions in NULL predicates is irrelevant since all operations involving a null value evaluate to the null value.

The NULL predicate is the only way to test for the presence of the null value in a column, since all other predicates where at least one of the operands is null evaluate to unknown.

The EXISTS Predicate

The EXISTS predicate tests whether the set of values addressed by a select-specification is empty or not, and has the form:



The result of the EXISTS predicate is true if the select-expression does not result in an empty set. Otherwise the result of the predicate is false. A set containing only null values is not empty. The result is never unknown.

The EXISTS predicate is the only predicate which does not compare a value with one or more other values. The columns selected in the select-expression of an EXISTS predicate are irrelevant. Most commonly, the SELECT * shorthand is used.

The EXISTS predicate may be negated in the construction of search conditions.

Examples

Consider the four following examples, and note particularly that the last example is true if all guests have undefined names:

Example 1

```
EXISTS (SELECT * FROM BOOK_GUEST
        WHERE GUEST = 'DATE')
```

requires that at least one guest is called DATE.

Example 2

```
NOT EXISTS (SELECT * FROM BOOK_GUEST
            WHERE GUEST = 'DATE')
```

requires that no guest may be called DATE.

Example 3

```
EXISTS (SELECT * FROM BOOK_GUEST
        WHERE NOT GUEST = 'DATE')
```

requires that at least one guest is not called DATE.

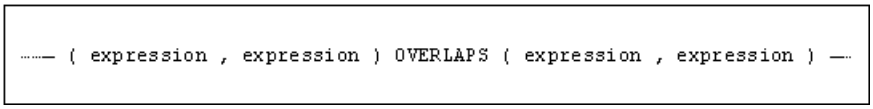
Example 4

```
NOT EXISTS (SELECT * FROM BOOK_GUEST
            WHERE NOT GUEST = 'DATE')
```

requires that no guest may not be called DATE, i.e. every guest must be called DATE (or be null).

The OVERLAPS Predicate

The OVERLAPS predicate tests whether two 'events' cover a common point in time or not, and has the form:



Each of the two events specified on either side of the OVERLAPS keyword is a period of time between two specified points on the time-line. The two points can be specified as a pair of datetime values or as one datetime value and an INTERVAL offset.

The first column in each row value expression must be a DATE, TIME or TIMESTAMP and the value in the first column of the first event must be comparable, see *Datetime Assignment Rules* on page *79*, to the value in the first column of the second event.

The second column in each row value expression may be either a DATE, TIME or TIMESTAMP that is comparable with the value in the first column or an INTERVAL with a precision that allows it to be added to the value in the first column.

The value in the first column of each row value expression defines one of the points on the time-line for the event.

If the value in the second column of the row value expression is a datetime, it defines the other point on the time-line for the event.

If the value in the second column of the row value expression is an `INTERVAL`, the other point on the time-line for the event is defined by adding the values in the two column of the row value to expression together.

Either of the two points may be the earlier point in time.

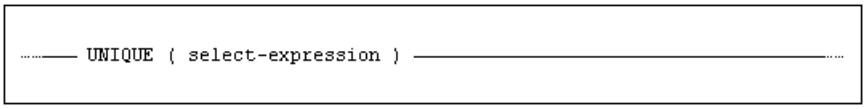
If the value in the first column of the row value expression is the null value, then this is assumed to be the later point in time.

The result of `(S1,T1) OVERLAPS (S2,T2)` is the result of the following expression:

```
(S1 > S2 AND NOT (S1 >= T2 AND T1 >= T2))
OR
(S2 > S1 AND NOT (S2 >= T1 AND T2 >= T1))
OR
(S1 = S2 AND (T1 <> T2 OR T1 = T2))
```

The UNIQUE Predicate

The `UNIQUE` predicate tests whether all rows returned by a `select-specification` are unique or not, and has the form:



The result of the `UNIQUE` predicate is true if the `select-expression` does not return any duplicates. Otherwise the result of the predicate is false. The result is never unknown. Null values are not considered equal to any values, including other null values.

The `UNIQUE` predicate may be negated in the construction of search conditions.

Examples

Return all artists that have only released one item:

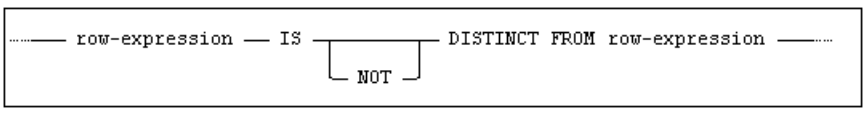
```
SELECT A.*
FROM MIMER_STORE.ARTISTS A
WHERE UNIQUE (SELECT ARTIST_ID
              FROM MIMER_STORE_MUSIC.TITLES T
              WHERE T.ARTIST_ID = A.ARTIST_ID)
```

Return all artists that have released items on different formats:

```
SELECT A.*
FROM MIMER_STORE.ARTISTS A
WHERE NOT UNIQUE (SELECT FORMAT
                  FROM MIMER_STORE_MUSIC.DETAILS D
                  WHERE D.ARTIST_ID = A.ARTIST_ID)
```

The DISTINCT Predicate

The `DISTINCT` predicate tests whether two values are distinct from each other or not, and has the form:



If both values are null, the result of the `DISTINCT` predicate is false. If only one of the values is null, the result of the predicate is true. If none of the values is null, the result of the predicate is true if the values are not the same.

This means that

```
x IS NOT DISTINCT FROM y
```

is equivalent to

```
x = y OR (x IS NULL AND y IS NULL)
```

And

```
x IS DISTINCT FROM y
```

is equivalent to

```
x <> y OR (x IS NULL AND y IS NOT NULL) OR (x IS NOT NULL AND y IS NULL)
```

Examples

The following examples are intended to show the difference between *distinct from* and *not equal to* when it comes to null values.

Select currencies that have an exchange rate distinct from Sweden's non-null exchange rate:

```
SELECT C1.*
FROM MIMER_STORE.CURRENCIES C1
JOIN MIMER_STORE.CURRENCIES C2
    ON C1.EXCHANGE_RATE IS DISTINCT FROM C2.EXCHANGE_RATE
WHERE C2.CODE = 'SEK'
```

The above query will return 161 rows. Countries having a null exchange rate are included.

Select currencies that have an exchange rate not equal to Sweden's non-null exchange rate:

```
SELECT c1.*
FROM MIMER_STORE.CURRENCIES C1
JOIN MIMER_STORE.CURRENCIES C2
    ON C1.EXCHANGE_RATE <> C2.EXCHANGE_RATE
WHERE C2.CODE = 'SEK'
```

The above query will return 154 rows. Countries having a null exchange rate are excluded.

Select currencies that have an exchange rate distinct from Saint Helena's null exchange rate:

```
SELECT C1.*
FROM MIMER_STORE.CURRENCIES C1
JOIN MIMER_STORE.CURRENCIES C2
    ON C1.EXCHANGE_RATE IS DISTINCT FROM C2.EXCHANGE_RATE
WHERE C2.CODE = 'SHP'
```

The above query will return 151 rows. No countries having a null exchange rate are included.

Select currencies that have an exchange rate not equal to Saint Helena's null exchange rate:

```
SELECT COUNT(*)
FROM MIMER_STORE.CURRENCIES C1
JOIN MIMER_STORE.CURRENCIES C2
    ON C1.EXCHANGE_RATE <> C2.EXCHANGE_RATE
WHERE C2.CODE = 'SHP'
```

The above query will return 0 rows.

Standard Compliance

This section summarizes standard compliance concerning predicates.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F052, "Intervals and datetime arithmetic" overlaps predicate. Feature F053, "OVERLAPS predicate". Feature F281, "LIKE enhancements". The arguments for a LIKE predicate may be value expressions. Feature F291, "UNIQUE predicate". Feature F561, "Full value expression". Feature F641, "Row and table constructors". Feature T022, "Advanced support for BINARY and VARBINARY data types". Feature T151, "DISTINCT predicate". Feature T152, "DISTINCT predicate with negation". Feature T461, "Symmetric BETWEEN predicate". Feature T501, "Enhanced EXISTS predicate".

Chapter 10

Search Conditions and Joins

This chapter discusses search conditions, that is, composite predicates defining row subsets from tables; and joined tables and inner and outer join specifications.

Search Conditions

Search conditions are used in:

- `WHERE` and `HAVING` clauses to qualify the selection of rows and groups respectively
- `CHECK` clauses to define sets of acceptable values
- `CASE` expressions to conditionally return different values
- `CASE` and `IF` statements to control conditional execution in a routine or trigger
- `WHILE` and `REPEAT` statements to control conditional iteration in a routine or trigger
- the `WHEN` clause of a trigger to control conditional execution of the trigger action.

A search condition is built from one or more predicates linked by the logical operators `AND` and `OR` and qualified if desired by the operator `NOT`.

A search condition is a boolean returning expression. (See *Expressions* on page *141*.)

Rules

Search conditions enclosed in parentheses may be used as part of more complex search condition constructions. A search condition is evaluated as follows:

- Conditions in parentheses are evaluated first.
- Within the same level of parentheses, `NOT` is applied before `AND`, `AND` is applied before `OR`.
- Operators at the same precedence level are applied in an order determined by internal optimization routines.

The result of a search condition is evaluated by combining the results of the component predicates. Each predicate evaluates to true, false or unknown, truth tables are shown in *Truth Tables* on page *82*.

WHERE and HAVING clauses select the set of values for which the search condition evaluates to true. CHECK clauses define the set of values for which the search condition does not evaluate to false, i.e. is either true or unknown.

Examples

Using WHERE

The WHERE condition determines which rows to select, for example:

```
SELECT *
FROM customer_details
WHERE country_code = 'SE'
```

Using HAVING

The HAVING clause restricts the selection of groups. A HAVING clause may contain set functions in the search condition. See *Mimer SQL User's Manual, Chapter 3, Grouped Set Functions – the GROUP BY Clause*.

Standard Compliance

This section summarizes standard compliance concerning search conditions.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Joined Tables

The JOIN syntax provides methods of combining information in tables.

INNER JOINs and OUTER JOINs are supported.

The different ways in which the various join options can be combined makes the overall JOIN syntax quite convoluted, so to simplify the explanation, each variant will be described on its own with an accompanying syntax diagram.

In order to understand the JOIN syntax generally, it is important to appreciate the difference between an INNER JOIN and an OUTER JOIN. It is also important to be aware of the difference between JOIN ON, JOIN USING and NATURAL JOIN.

INNER JOINs

An INNER JOIN produces a result table containing composite rows created by combining rows from two tables where some pre-defined, or explicitly specified, join condition evaluates to true.

Rows that do not satisfy the JOIN condition will not appear in the result table of an INNER JOIN.

The INNER JOIN is the default JOIN type, and the INNER keyword is optional and may be omitted.

JOIN ON

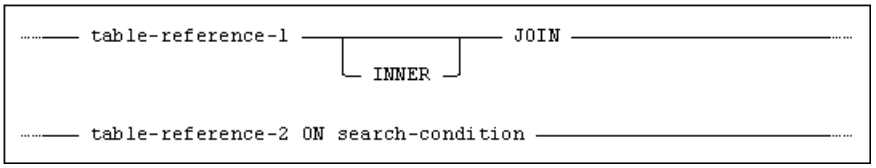
JOIN ON allows a join condition to be specified. The result table of this kind of join is produced by applying the specified join condition to the Cartesian product of the two tables. The result table will contain only those rows for which the join condition evaluates to true.

The join condition cannot reference common columns unless they are qualified (e.g. by table name).

A row in the result table contains the combined set of columns from each table. The columns from table-reference-1 appear first followed by those from table-reference-2. Common columns will therefore appear twice.

Syntax

The syntax for JOIN ON is:



Example

The ON condition is used to select the employees' corresponding row(s) salaries, and the WHERE condition is used to find those whose salaries are less than 10 000.

```
SELECT *
FROM epmloyees INNER JOIN salaries ON epmloyees.id = salaries.id
WHERE salaries.salary < 10000
```

JOIN USING

`JOIN USING` allows a list of common column names to be specified. The result contains one row for each case where all the specified columns in the two tables contain values that are equal. (This kind of join is conceptually the same as a `NATURAL JOIN` except that the join is based on the specified columns rather than on all the common columns.)

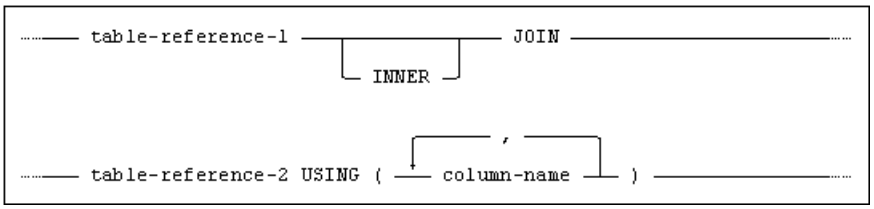
Specifying the columns explicitly instead of using the entire set of common columns is useful in situations where some of the common columns may not contain identical values even though the respective rows are related (e.g. as in a `REMARKS` column).

The columns specified in the list must be common to both tables, they must be specified in an unqualified manner and must have a data type that allows the values in the respective tables to be compared.

A row in the result table contains the combined set of columns from each table, except that the common columns appear only once. The columns specified in the list of column names appear first (at the left of the table) followed by the remaining columns from `table-reference-1`, followed by those from `table-reference-2`.

Syntax

The syntax for `JOIN USING` is:



Example

The construction below will make a join on the specified column `id`.

```
SELECT * FROM epmloyees JOIN salaries USING (id)
```

NATURAL JOIN

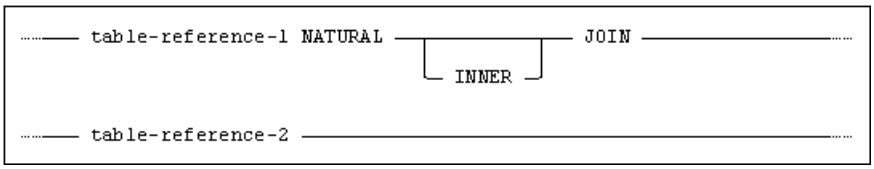
The result table of a `NATURAL JOIN` contains one row for each case where all the common columns in the two tables contain values that are equal.

Common columns are those which have the same name in each table. The common columns must have a data type that allows values in the respective tables to be compared.

A row in the result table contains the combined set of columns from each table, except that the common columns appear only once. The common columns appear first (at the left of the table) followed by the remaining columns from `table-reference-1`, followed by those from `table-reference-2`.

Syntax

The syntax for a NATURAL JOIN is:



If there are no rows where all the common columns have equal values, the result table is an empty table (i.e. it has a set of columns as just described, but the number of rows is zero).

For example, two tables contain different sets of information on people, and each table has a FIRST_NAME column and a SURNAME column to identify the person to whom the information applies.

When both the FIRST_NAME column and the SURNAME column contain the same values in a row in each table, it means those rows are related. A NATURAL JOIN between these two tables would produce a result table with a single composite row for each person, containing all the information held in both tables, with the SURNAME and FIRST_NAME columns appearing once in the rows of the result.

Note: It is actually possible to perform a NATURAL JOIN between two tables which have no common columns at all. In this case the result table is the Cartesian product (sometimes called the CROSS JOIN) of the two tables.

Examples

The NATURAL INNER JOIN construction below will join the employees and salaries tables on all columns which share the same name.

```
SELECT * FROM employees NATURAL JOIN salaries
```

Note: Use of NATURAL JOIN is discouraged in programs since subsequent alterations in the table or view definitions may result in an additional join condition (e.g. if a column with the same name as the new column already exists in the other table), and cause the program to function incorrectly.

OUTER JOINs

A table resulting from an inner join, as just described, will only contain those rows that satisfy the applicable join condition. This means that a row in either table which does not match a row in the other table will be excluded from the result.

In an OUTER JOIN, however, a row that does not match a row in the other table is also included in the result table. Such a row appears once in the result and the columns that would normally contain information from the other table will contain the null value.

The join variants (JOIN ON, NATURAL JOIN and JOIN USING) can be applied as OUTER JOINs as well.

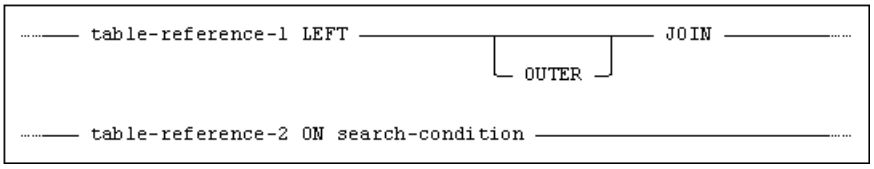
The OUTER keyword is optional and may be omitted.

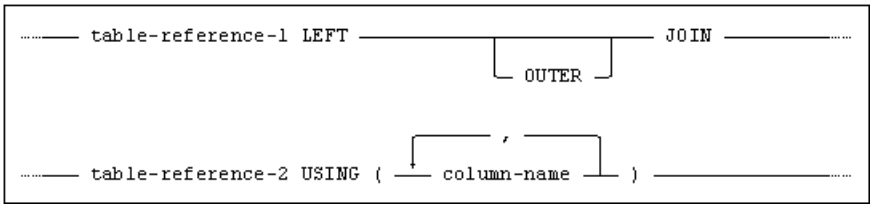
LEFT OUTER JOIN

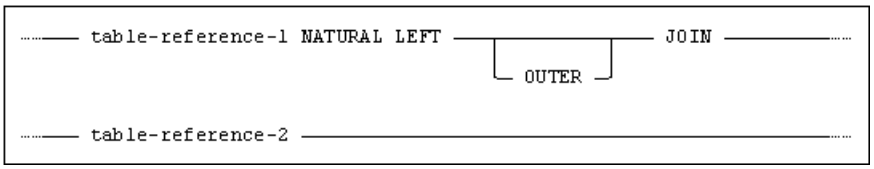
In addition to the `INNER JOIN` result, the `LEFT OUTER JOIN` also includes the rows from `table-reference-1` (the table on the left of the `JOIN`) which do not satisfy the join condition.

Syntax

The syntax variants of the `LEFT OUTER JOIN` are as follows:







Example

The query below will return a result set containing all employees at least once even though they might not have an entry in the `SALARIES` table.

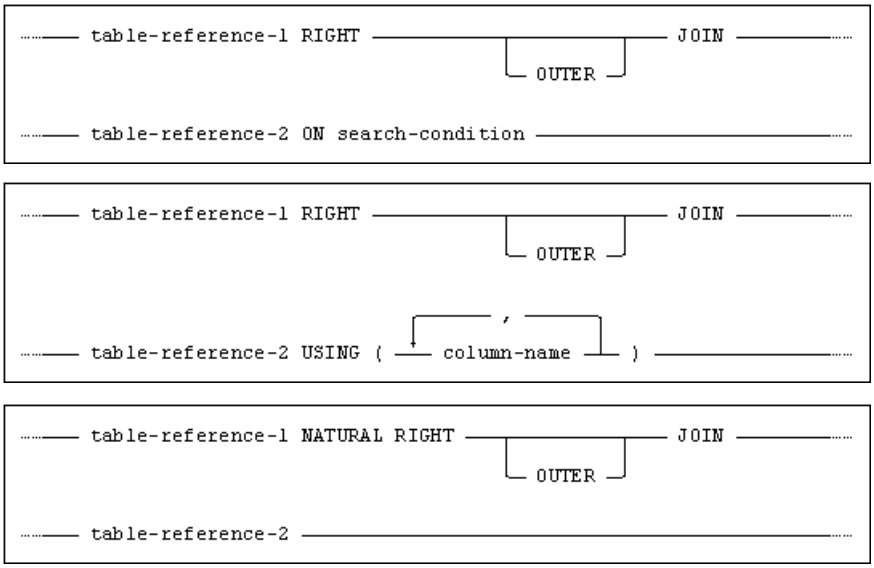
```
SELECT *
FROM employees
LEFT JOIN salaries
    ON employees.id = salaries.id
```

RIGHT OUTER JOIN

In addition to the `INNER JOIN` result, the `RIGHT OUTER JOIN` also includes the rows from `table-reference-2` (the table on the right of the `JOIN`) which do not satisfy the join condition.

Syntax

The syntax for the variants of the `RIGHT OUTER JOIN` is as follows:



Example

The query below counts the number of releases for each month during the year 1990, using `RIGHT OUTER JOIN` to include months without any release.

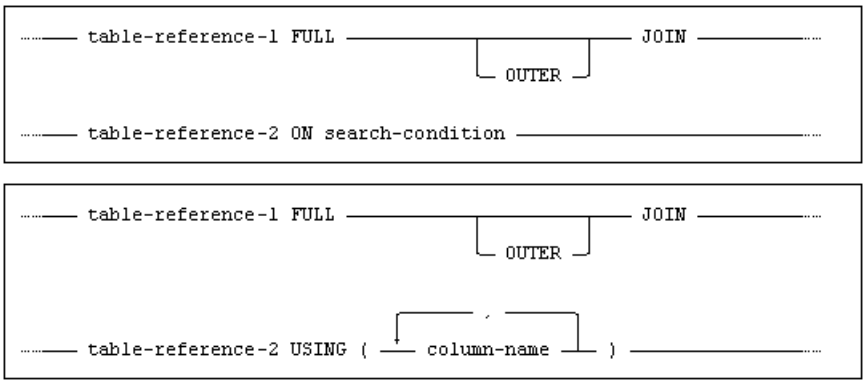
```
select smr.c as month_no, count(i.item_id) as number_of_releases
from items i
right join system.manyrows smr
    on  extract(month from i.release_date) = smr.c
    and extract(year from i.release_date) = 1990
where smr.c between 1 and 12
group by smr.c
```

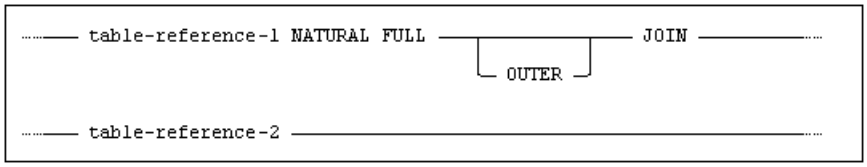
FULL OUTER JOIN

A `FULL OUTER JOIN` combines the effect of both a `LEFT JOIN` and a `RIGHT JOIN`, i.e. the `FULL OUTER JOIN` includes the rows from the left table which do not satisfy the join condition, and the rows from the right table which do not satisfy the join condition.

Syntax

The syntax for the variants of the `FULL OUTER JOIN` is as follows:





Example

A full outer join of the SELLERS table with the SUPPLIERS table on the CITY column:

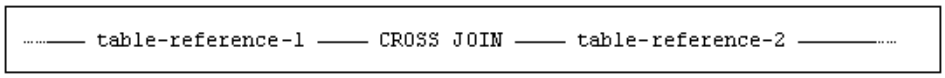
```
SELECT sellers.seller_name, suppliers.supplier_name
FROM sellers
FULL OUTER JOIN suppliers
    ON sellers.city = suppliers.city
```

CROSS JOIN

A CROSS JOIN is basically an INNER JOIN between two tables without a join condition. The result table is the Cartesian product of the two tables.

Syntax

The syntax for CROSS JOIN is as follows:



Example

```
SELECT * FROM
(SELECT COUNT(*) AS store_count FROM stores) s
CROSS JOIN
(SELECT COUNT(*) AS emp_count FROM employees) e
```

Standard Compliance

This section summarizes standard compliance concerning JOIN.

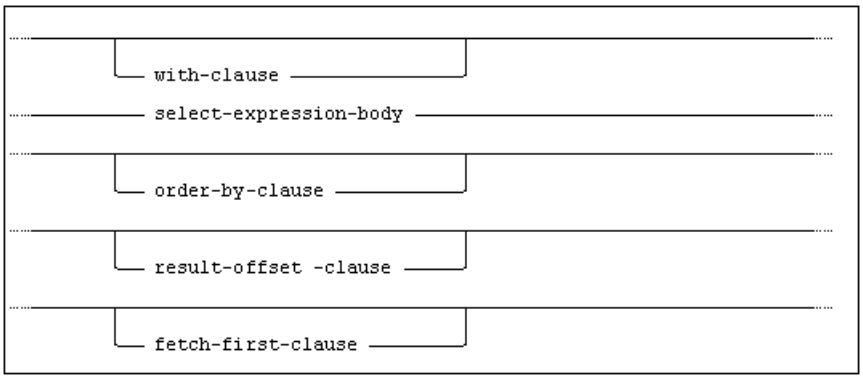
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F401 “Extended joined table”, NATURAL JOIN, FULL OUTER JOIN and CROSS JOIN.

Chapter 11

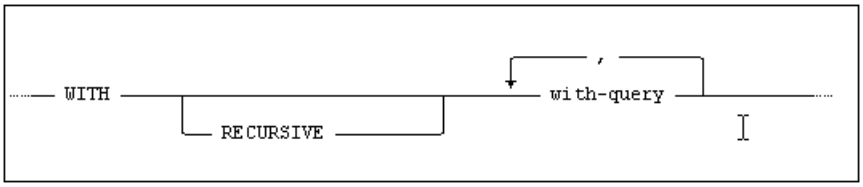
The SELECT Expression

A `select-expression` defines a set of data (rows and columns) extracted from one or more tables or views.

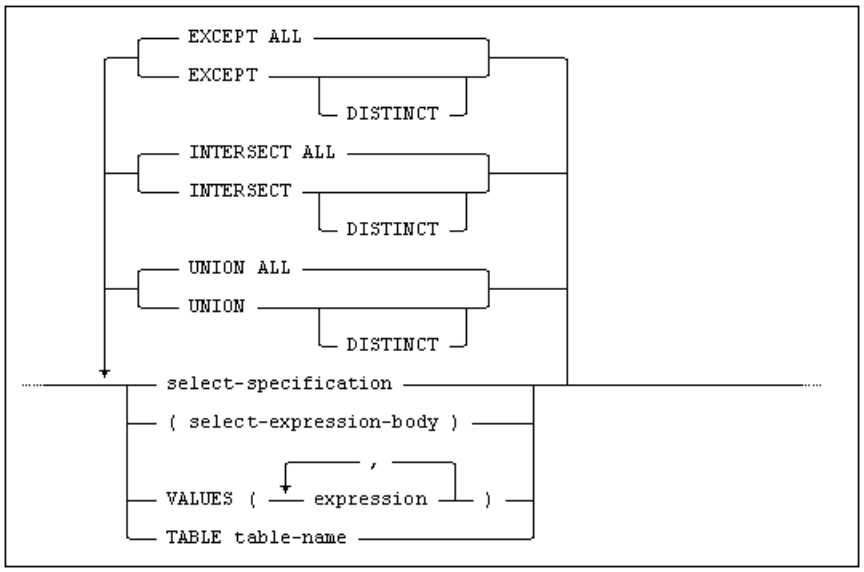
The `select-expression` syntax is:



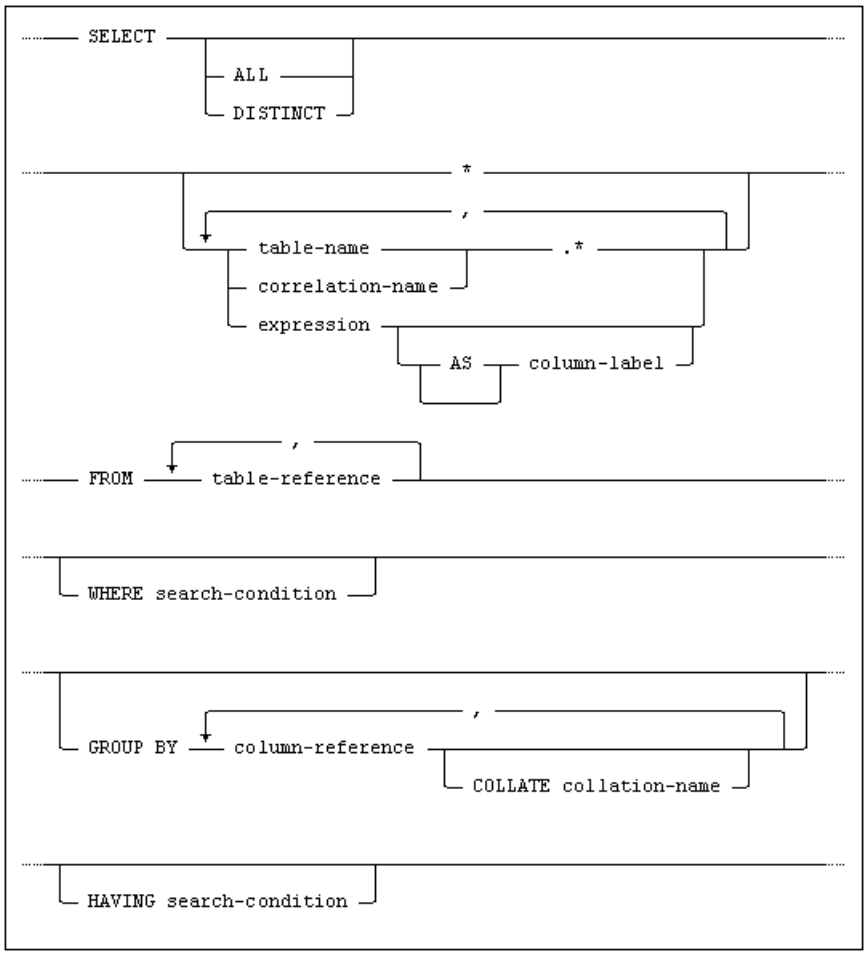
where `with-clause` is:



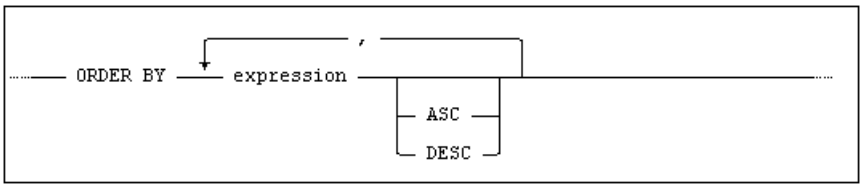
and `select-expression-body` is:



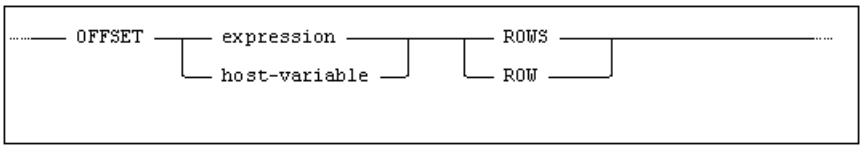
where the `select-specification` syntax is:



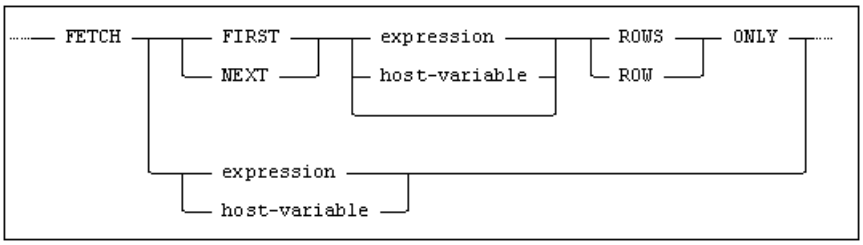
and `order-by-clause` is:



and `result-offset-clause` is:



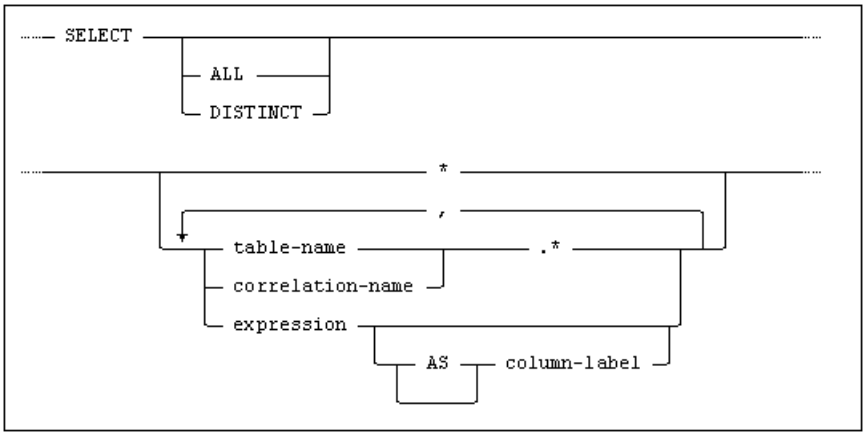
and `fetch-first-clause` is:



The different clauses in the specifications above are described in detail in the following sections.

The SELECT Clause

The `SELECT` clause defines which values are to be selected. Values are specified by column references or expressions; where columns are addressed, the value selected is the content of the column.



SELECT *

This form of the `SELECT` clause specifies all columns from the `FROM` clause. The single asterisk may not be combined with any other value specification.

Example

```
SELECT * FROM countries ...
```

Note: Use of `SELECT *` is discouraged in programs (except in `EXISTS` predicates) since the asterisk is expanded to a column list when the statement is compiled, and any subsequent alterations in the table or view definitions may cause the program to function incorrectly.

SELECT table.*

If a named table or view (`table-name` or `correlation-name`) is followed by an asterisk in the `SELECT` clause, all columns are selected from that table or view.

This formulation may be used in a list of select specifications.

If a `correlation-name` is used, it must be defined in the associated `FROM` clause, see *The FROM Clause and Table-reference* on page *177*.

Note: Use of `SELECT table.*` is discouraged in programs since the asterisk is expanded to a column list when the statement is compiled, and any subsequent alterations in the table or view definitions may cause the program to function incorrectly.

SELECT expression

Values to be selected may be specified as expressions (using column-references, set functions and literals, see *Expressions* on page *141*).

Column names used in expressions must refer to columns in the tables addressed in the `FROM` clause, or be an outer reference.

A column name must be qualified if more than one column in the set of table references addressed in the `FROM` clause has the same name.

SELECT … AS Column-label

A `column-label` may be added after each separate expression in the `SELECT` clause. `column-label` is an SQL identifier which becomes the name of the column in the result set.

If no name is given the original column name is used, unless the new column was created by an expression, in which case the new column has no name.

For example, `SELECT COLUMN_NAME` would result in a column called `COLUMN_NAME` in the result set, but `SELECT COLUMN_NAME + 1` would result in a column in the result set with no name.

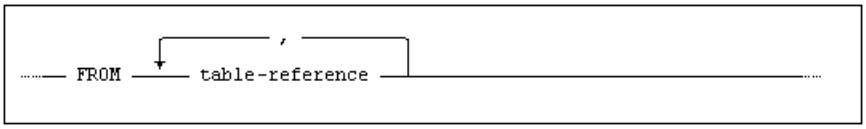
The Keywords ALL and DISTINCT

If `ALL` is specified or if no keyword is given, duplicate rows are not eliminated from the result of the `select-specification`.

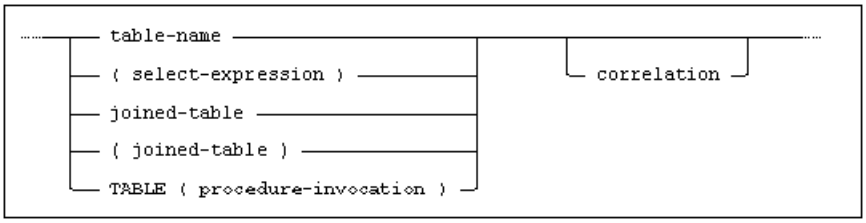
If `DISTINCT` is specified, duplicate rows are eliminated. Null is considered to be equal to null in this context.

The FROM Clause and Table-reference

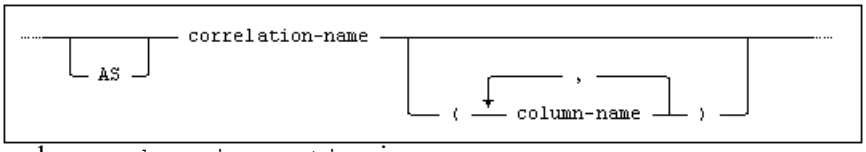
The FROM clause defines an intermediate result set for the select-specification, and may define correlation names for the table references used in the result set.



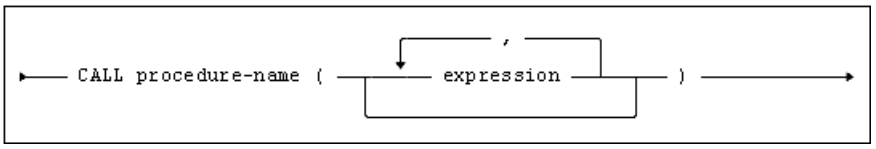
where table-reference is:



where correlation is:



and procedure-invocation is:



General Syntax

All source tables or views referenced in the SELECT clause and at the top level in the WHERE clause (but not in any subquery used in the WHERE clause) must be named in the FROM clause.

Intermediate Result Sets

If a single table or view is named in the FROM clause, the intermediate result set is identical to the table or view.

If the FROM clause names more than one table or view, the intermediate result set may be regarded as the complete Cartesian product of the named tables or views.

Note: The intermediate result set is a conceptual entity, introduced to aid in understanding of the selection process. The complete result set does not have any direct physical existence, so that the machine resources available do not need to correspond to the (sometimes very large) Cartesian product tables implied by multiple table references in a FROM clause.

Correlation Names

Correlation names introduced in the FROM clause redefine the form of the table name which may be used to qualify column names, see *Qualified Object Names* on page *39*.

Correlation names may be used for several purposes:

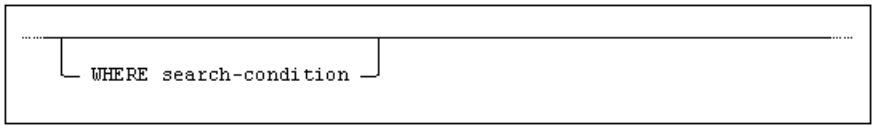
- to shorten table names, which saves typing and makes statements easier to follow and less error-prone.
- to relate a table to a logical copy of itself.
- to rename a column when a column with the same name exists in another of the query's tables.

A table or view name is exposed in the FROM clause if it does not have a correlation name. The same table or view name cannot be exposed more than once in the same FROM clause.

The same correlation name may not be introduced more than once in the same FROM clause, and it cannot be the same as an exposed table or view name.

The WHERE Clause

The WHERE clause selects a subset of the rows in the intermediate result set on the basis of values in the columns. If no WHERE clause is specified, all rows of the intermediate result set are selected.



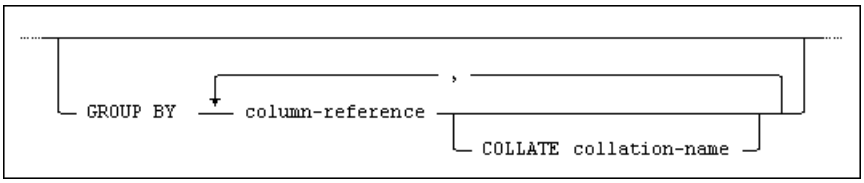
All column references in the search-condition must uniquely identify a column in the intermediate result set defined by the FROM clause or be an outer reference.

Column references must be qualified if more than one column in the intermediate result set has the same name, or if the column is an outer reference.

The GROUP BY Clause

The GROUP BY clause determines grouping of the result table for the application of set functions specified in the SELECT clause.

The GROUP BY clause has the following syntax:



If a GROUP BY clause is specified, each column reference in the SELECT list must either identify a grouping column or be the argument of a set function.

The rows of the intermediate result set are (conceptually) arranged in groups, where all values in the grouping column(s) are identical within each group.

Each group is reduced to a single row in the final result of the select-specification.

If a GROUP BY clause is not specified, the SELECT list must either be a list that does not include any set functions or a list of set functions and optional literal expressions.

The COLLATE Clause

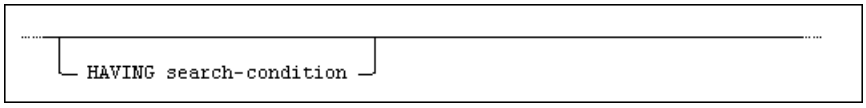
Collations determine the sort order of character data. If the `COLLATE` clause is specified, the resulting data will be grouped according to the collation specified. For more information, see the *Mimer SQL User's Manual, Chapter 4, Collations*.

If no `COLLATE` clause is specified, the column's implicit collation will be used.

The HAVING Clause

The `HAVING` clause restricts selection of groups in the same way that a `WHERE` clause restricts selection of rows.

The `HAVING` clause has the following syntax:



The search condition in the `HAVING` clause defines restrictions on the values in the elements of the `SELECT` list. Column references in the search condition of the `HAVING` clause must identify a grouping column, or be used in set functions, or be outer references.

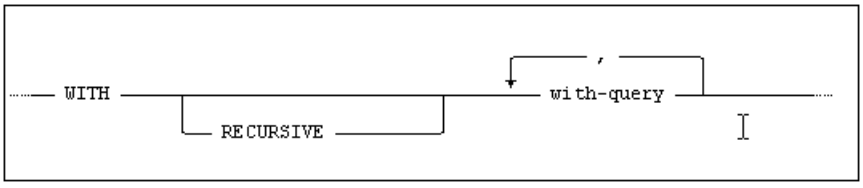
Most commonly, `HAVING` is used together with `GROUP BY`, in which case the search conditions relate either values in grouping columns or results of set functions to expressions.

If the `HAVING` clause is used without a `GROUP BY` clause, all rows in the result table are treated as a single group. In this case, the `HAVING` clause must refer to a set function (since there are no grouping columns).

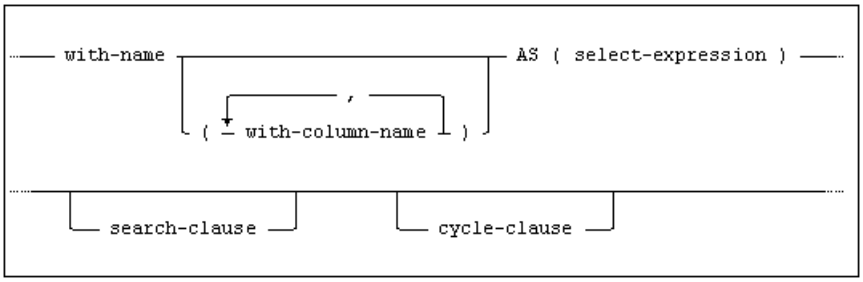
The WITH Clause

A `WITH` clause provides a way to write named queries for use in a larger query. (Such a named query is also known as a cte, common table expression.)

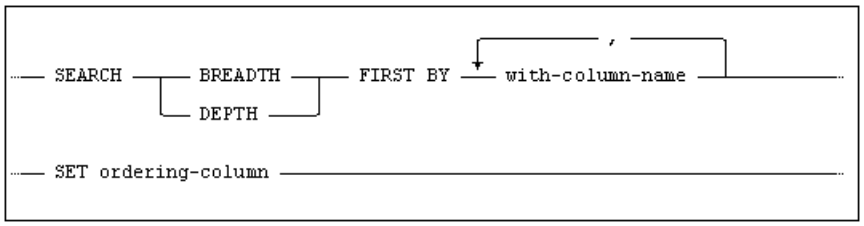
The `WITH` clause has the following syntax:



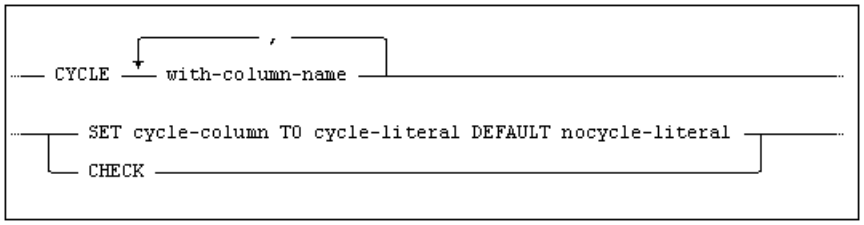
where `with-query` is:



where `search-clause` is:



where `cycle-clause` is:



The `WITH` clause contains one or more named select expressions that can be referenced multiple times in the query following the `WITH` clause. These select expressions can be seen as a temporary views, which are only defined within a query.

`WITH` clauses can be nested, i.e. a `SELECT` expression in a with element or a subquery may also contain with clauses. These with elements are not in scope outside of the context in which they are defined.

If the column names are omitted from a with element, the names from the outermost select list will be used.

Example

```
WITH display_order (display_order, format) AS
(
    SELECT CASE display_order
                WHEN 10 THEN 'FIRST'
                WHEN 20 THEN 'SECOND'
                WHEN 30 THEN 'THIRD'
                WHEN 40 THEN 'FOURTH'
                ELSE 'UNKNOWN'
           END,
           format
    FROM formats
)
SELECT display_order,
       format
FROM display_order
WHERE display_order IN ('SECOND', 'THIRD')
```

The usage of a `WITH` clause in the previous example can also be expressed by using a derived table and thus this use of `WITH` does not add any new functionality. However, a query written in this way can be found easier to construct and to read.

Example

```
WITH annualsalary AS
(
    select staff_id, salyear, sum(payment) as salary
    from (select staff_id, payment, extract(year from paymentdate) as salyear
          from payments) p
    group by staff_id, salyear
)
select emp.name as emp_name, esal.salary as emp_sal,
       mngr.name as mngr_name, msal.salary as mngr_sal, esal.salyear
from staff emp
join annualsalary esal
    on emp.id = esal.staff_id
left join staff mngr
    on emp.manager_id = mngr.id
left join annualsalary msal
    on mngr.id = msal.staff_id and esal.salyear = msal.salyear
```

The `WITH` clause above specifies the named query `annualsalary`, which is joined twice in the main query. Writing this query only once makes coding more efficient, and future changes will be simpler and safer.

Recursive Queries

A `WITH` clause query that refers to its own output is recursive. Recursive queries make it possible to express things otherwise not possible using a single SQL statement.

The general form of a recursive `WITH` query is always a non-recursive term, the *anchor member*, with a `UNION` (or `UNION ALL`), followed by a query, the *recursive member*, which contains a reference to the with element (*anchor member*) itself.

A simple example is to generate the integer values from 1 to 50:

```
WITH RECURSIVE integer_list(n) as
(
    VALUES (1)          -- anchor member

    UNION ALL

    SELECT n + 1
    FROM integer_list   -- recursive member
    WHERE n < 50        -- terminating condition
)
select n
from integer_list
```

Recursive queries are typically used to deal with hierarchical or tree-structured data, e.g. solving the bill of materials problem (see https://en.wikipedia.org/wiki/Bill_of_materials.)

Another example is to show categories with sub-categories and where a sub-category may have sub-categories by its own and so on indefinitely:

```
with recursive tree (name, parent, level) as
(
    select name, parent, 0
    from categories
    where parent is null
    union all
    select categories.name, categories.parent, level + 1
    from tree
        join categories on tree.name = categories.parent
)
select *
from tree
order by level
```

When working with recursive queries it is important to be sure that the recursive part of the query has a terminating condition as the query otherwise will not terminate.

BREADTH FIRST and DEPTH FIRST

When a `SEARCH` clause is specified the ordering-column is added to the result set of the `WITH` clause. It is set to a sequence of values that reflects `BREADTH` or `DEPTH` first traversal of the recursive query. The traversal is done according the search columns specified for each level in the tree. Even though search order is specified the rows may be returned in any order, so if you want the rows actually returned in this order you must add an `ORDER BY` ordering-column to your query.

A `SEARCH` clause may only be specified with a recursive `WITH` clause.

Example

Return combined values, using the different traversal methods.

```
create table t (c varchar(1));
insert into t values ('A');
insert into t values ('B');
insert into t values ('C');

SQL>WITH tr (grp, c) AS
SQL&(
SQL&  SELECT cast(c as varchar(20)), c
SQL&  FROM t
SQL&
SQL&  UNION ALL
SQL&
SQL&  SELECT c.grp || ', ' || t.c, t.c
SQL&  FROM t
SQL&  INNER JOIN tr c
SQL&    ON c.c < t.c
SQL&) SEARCH BREADTH FIRST BY grp SET oc
SQL&SELECT grp FROM tr;
grp
==========================
A
B
C
A, B
A, C
B, C
A, B, C

                  7 rows found

SQL>
SQL>WITH tr (grp, c) AS
SQL&(
SQL&  SELECT cast(c as varchar(20)), c
SQL&  FROM t
SQL&
SQL&  UNION ALL
SQL&
SQL&  SELECT c.grp || ', ' || t.c, t.c
SQL&  FROM t
SQL&  INNER JOIN tr c
SQL&    ON c.c < t.c
SQL&) SEARCH DEPTH FIRST BY grp SET oc
SQL&SELECT grp FROM tr;
grp
==========================
A
A, B
A, B, C
A, C
B
B, C
C

                  7 rows found
```

Cycle detection

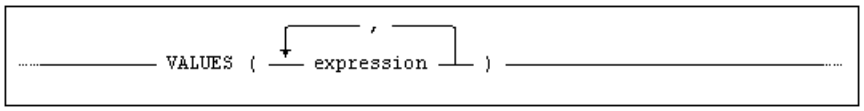
A CYCLE clause can be specified when the traversal may contain cycles. If there is no cycle checking a result can return an infinite number of rows. If a cycle-column is specified, it is added to the result set of the WITH clause. When a cycle is detected, the row with the cycle is returned and the cycle-column is set to the cycle-literal. It will then continue with the next row in the tree without following the cycle. If the cycle-check is specified an error code (-12288) will be returned when a cycle is detected and execution stops.

A CYCLE clause may only be specified with a recursive WITH clause.

The VALUES Clause

VALUES computes a row value specified by value expressions.

The VALUES clause has the following syntax:



To get multiple row values several VALUES clauses can be unioned together. It is most commonly used to generate a "constant table" within a larger command, but it can be used on its own.

Example

```
select *
from
(
    values('A', 1)
    union
    values('B', 2)
) dt(x, y)
```

will return two rows with the columns named x and y.

The UNION Operator

If several SELECT statements are connected by UNION (or UNION DISTINCT), the result is derived by first merging all result tables specified by the separate SELECT statements, and then eliminating duplicate rows from the merged set. All columns in the result table are significant for the purpose of eliminating duplicates.

The UNION ALL operator on the other hand retains all duplicates. The operator can be viewed as a way to concatenate several queries.

The rules described below apply to both UNION and UNION ALL.

All separate result tables from SELECT statements connected by UNION must have the same number of columns and the data types of columns to be merged must be compatible.

The columns in the result table are named in accordance with the columns in the first SELECT statement of the UNION construction.

Separate SELECT statements may be enclosed in parentheses if desired. This does not affect the result of a UNION operation.

The names in the first select specification are used in UNION constructions.

See *Result Data Types* on page *84* for a description of how the data type of the UNION result is determined.

The EXCEPT Operator

If several SELECT statements are connected by EXCEPT (or EXCEPT DISTINCT), the result is derived by taking the distinct rows of the first query and return the rows that do not appear in second result query. All columns in the result table are significant for the purpose of eliminating duplicates.

The EXCEPT ALL operator on the other hand retains all duplicates.

The rules described below apply to both EXCEPT and EXCEPT ALL.

All separate result tables from SELECT statements connected by EXCEPT must have the same number of columns and the data types of columns to be merged must be compatible.

The columns in the result table are named in accordance with the columns in the first SELECT statement of the EXCEPT construction.

Separate SELECT statements may be enclosed in parentheses if desired. This does not affect the result of a EXCEPT operation.

The names in the first select specification are used in EXCEPT constructions.

See *Result Data Types* on page *84* for a description of how the data type of the EXCEPT result is determined.

The INTERSECT Operator

If several SELECT statements are connected by INTERSECT (or INTERSECT DISTINCT), the result is derived by taking the results of two queries and return only rows that appear in both result sets, and then eliminating duplicate rows from the merged set. All columns in the result table are significant for the purpose of eliminating duplicates.

The INTERSECT ALL operator on the other hand retains all duplicates.

The rules described below apply to both INTERSECT and INTERSECT ALL.

All separate result tables from SELECT statements connected by INTERSECT must have the same number of columns and the data types of columns to be merged must be compatible.

The columns in the result table are named in accordance with the columns in the first SELECT statement of the INTERSECT construction.

Separate SELECT statements may be enclosed in parentheses if desired. This does not affect the result of a INTERSECT operation.

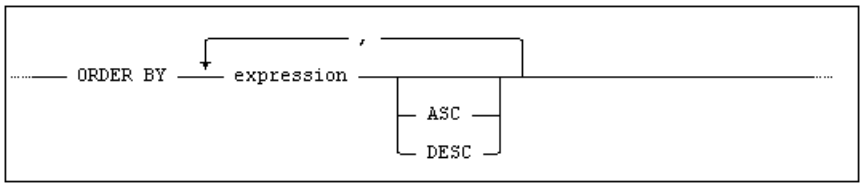
The names in the first select specification are used in INTERSECT constructions.

See *Result Data Types* on page *84* for a description of how the data type of the INTERSECT result is determined.

The ORDER BY Clause

The result table may be ordered according to an `order-by-clause`.

The `ORDER BY` clause has the following syntax:



Every expression in the `order-by-clause` must contain a reference to a column in a table specified in the `FROM` clause.

Column labels, created with `SELECT AS`, may not be part of a complex `ORDER BY` expression, (i.e. if column label is used, the expression must contain nothing but the column label).

The `ORDER BY` expressions must not include set functions (i.e. `MAX`, `MIN`, `AVG`, `SUM` and `COUNT`), subqueries or `NEXT VALUE FOR` sequence.

If `DISTINCT`, `GROUP BY`, `UNION`, `EXCEPT` or `INTERSECT` is specified, only columns from the result set may be specified as `ORDER BY` expressions.

The default collation for sorting data is the collation defined for the column being sorted. If you include a `COLLATE` clause, you can override the default collation by explicitly specifying a different collation. For more information, see the *Mimer SQL User's Manual, Chapter 4, Collations*.

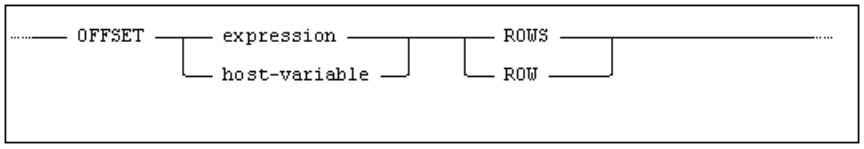
Ascending/Descending

For each column in the `order-by-clause`, the sort order may be specified as `ASC` (ascending) – the default, or `DESC` (descending). If more than one column is specified, the result table is ordered first by values in the first specified column, then by values in the second, and so on.

The RESULT OFFSET Clause

The `result-offset-clause` is used to limit the result set by removing a specified number of rows from its beginning.

The `result-offset-clause` clause has the following syntax:

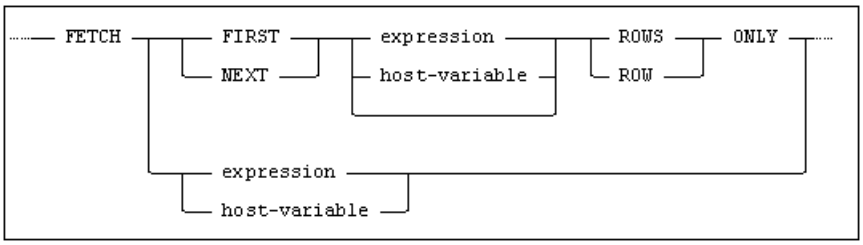


If a statement contains both an `order-by-clause` and a `result-offset-clause`, the result set is first sorted according to the `ORDER BY` clause, and then the number of rows specified in the `result-offset-clause` are removed.

The FETCH FIRST Clause

The `fetch-first-clause` is used to limit the result set by specifying the number of rows to be returned.

The `fetch-first-clause` has the following syntax:



If a statement contains both an `order-by-clause` and a `fetch-first-clause`, the result set is first sorted according to the `order-by-clause` and then limited to the number of rows specified in the `fetch-first-clause`.

If both a `result-offset-clause` and a `fetch-first-clause` are specified, the `result-offset-clause` is applied first, then the `fetch-first-clause`.

Restrictions

`SELECT` access is required on all tables and views specified in a `FROM` clause.

Notes

If the `SELECT` statement is used without the `ORDER BY` clause, the sort order is undefined. This means that the sort order may change if new indexes are created, indexes are dropped, new statistics are gathered or if a new version of the SQL optimizer is installed.

Standard Compliance

This section summarizes standard compliance for `select-specifications`.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F302, “INTERSECT table operator”. Feature F304, “EXCEPT ALL table operator”. Feature T551, “Optional keywords for default syntax” support for the keyword DISTINCT. Feature F591, “Derived tables”. Feature F661, “Simple tables” Feature F851, “<order by clause> in subqueries”. Feature F855, “Nested <order by clause> in <query expression>”. Feature F856,” Nested <fetch first clause> in <query expression>”. Feature F857, “Top-level <fetch first clause> in <query expression>” Feature F858, “<fetch first clause> in subqueries”. Feature F860, “dynamic <fetch first row count> in <fetch first clause>”. Feature F861, “Top-level <result offset clause> in <query expression>”. Feature F862, “<result offset clause> in subqueries”. Feature F863, “Nested <result offset clause> in <query expression>”. Feature F865, “dynamic <offset row count> in <result offset clause>”. Feature T121, “WITH (excluding RECURSIVE) in query expression” Feature T122, “WITH (excluding RECURSIVE) in subquery” Feature T131, “Recursive query” Feature T132, “Recursive query in subquery” Feature T551, “Optional key words for default syntax”.

Standard	Compliance	Comments
	Mimer SQL Extension	Support for host variable in <fetch first clause> and <result offset clause> is a Mimer SQL extension. <cycle literal> and <nocycle literal> must be data type CHAR(1) according to the SQL standard. Mimer SQL requires both literals to be of compatible types and can be boolean, numeric, or character. The option to perform CYCLE CHECK with an error code is a Mimer SQL extension.

Chapter 12

SQL Statements

This chapter documents SQL statements in Mimer SQL.

In SQL there are different types of statements:

- Procedural statements, including DML (Data Manipulation Language), see *Procedural SQL Statements* on page *193*.
- Data definition (DDL) statements, see *Data Definition Statements* on page *192*.
- Access control statements, see *Access Control Statements* on page *191*.
- Connection statements, see *Connection Statements* on page *191*.
- Declarative statements, see *Declarative Statements* on page *193*.
- Embedded SQL statements, see *Embedded SQL Statements* on page *193*.
- Embedded SQL control statements, see *Embedded SQL Control Statements* on page *193*.
- System administration statements, see *System Administration Statements* on page *194*.

Access Control Statements

Access control statements can be divided into two categories: `GRANT` and `REVOKE`.

For information on `GRANT` statements, see:

- *GRANT ACCESS PRIVILEGE* on page *359*
- *GRANT OBJECT PRIVILEGE* on page *361*
- *GRANT SYSTEM PRIVILEGE* on page *364*.

For information on `REVOKE` statements, see:

- *REVOKE ACCESS PRIVILEGE* on page *388*
- *REVOKE OBJECT PRIVILEGE* on page *391*
- *REVOKE SYSTEM PRIVILEGE* on page *394*.

Connection Statements

For information on connection statements, see:

- *CONNECT* on page *245*
- *DISCONNECT* on page *325*
- *ENTER* on page *331*

- *LEAVE (PROGRAM ident)* on page *375*
- *SET CONNECTION* on page *406*.

Data Definition Statements

For information on data definition (DDL) statements, see:

- *ALTER DATABANK* on page *200*
- *ALTER DATABASE* on page *207*
- *ALTER FUNCTION* on page *209*
- *ALTER IDENT* on page *212*
- *ALTER METHOD* on page *214*
- *ALTER PROCEDURE* on page *216*
- *ALTER ROUTINE* on page *219*
- *ALTER SEQUENCE* on page *222*
- *ALTER SHADOW* on page *223*
- *ALTER STATEMENT* on page *225*
- *ALTER TABLE* on page *226*
- *ALTER TYPE* on page *230*
- *COMMENT* on page *239*
- *CREATE COLLATION* on page *251*
- *CREATE DATABANK* on page *253*
- *CREATE DOMAIN* on page *256*
- *CREATE FUNCTION* on page *258*
- *CREATE IDENT* on page *262*
- *CREATE INDEX* on page *264*
- *CREATE MODULE* on page *269*
- *CREATE PROCEDURE* on page *271*
- *CREATE SCHEMA* on page *275*
- *CREATE SEQUENCE* on page *277*
- *CREATE SHADOW* on page *280*
- *CREATE STATEMENT* on page *282*
- *CREATE SYNONYM* on page *284*
- *CREATE TABLE* on page *285*
- *CREATE TRIGGER* on page *294*
- *CREATE VIEW* on page *302*
- *DROP* on page *326*.

Declarative Statements

Declarative statements, denoted as `declarative-statement` in syntax diagrams, include the following statements:

- *DECLARE CONDITION*, see page *307*
- *DECLARE CURSOR*, see page *309*
- *DECLARE HANDLER*, see page *312*
- *DECLARE VARIABLE*, see page *315*.

Embedded SQL Statements

Embedded SQL statements include the following statements:

- *ALLOCATE CURSOR*, see page *196*
- *ALLOCATE DESCRIPTOR*, see page *198*
- *DEALLOCATE DESCRIPTOR*, see page *305*
- *DEALLOCATE PREPARE*, see page *306*
- *DESCRIBE*, see page *323*
- *EXECUTE*, see page *332*
- *EXECUTE IMMEDIATE*, see page *334*
- *GET DESCRIPTOR*, see page *344*
- *PREPARE*, see page *380*
- *SET DESCRIPTOR*, see page *411*.

Embedded SQL Control Statements

Embedded SQL control statements include the following statements:

- *DECLARE SECTION*, see page *314*
- *WHENEVER*, see page *434*.

Procedural SQL Statements

Procedural SQL statements (including DML), denoted `procedural-sql-statement` in syntax diagrams, include the following statements:

- *CALL*, see page *233*
- *CASE*, see page *235*
- *CLOSE*, see page *237*
- *COMMIT*, see page *241*
- *COMPOUND STATEMENT*, see page *243*
- *DELETE CURRENT*, see page *319*
- *DELETE*, see page *317*
- *EXECUTE STATEMENT*, see page *335*
- *FETCH*, see page *339*

- *FOR*, see page *342*
- *GET DIAGNOSTICS*, see page *351*
- *IF*, see page *366*
- *INSERT*, see page *368*
- *ITERATE*, see page *371*
- *LEAVE*, see page *373*
- *LOOP*, see page *376*
- *OPEN*, see page *378*
- *REPEAT*, see page *382*
- *RESIGNAL*, see page *384*
- *RETURN*, see page *386*
- *ROLLBACK*, see page *396*
- *SELECT INTO*, see page *401*
- *SELECT*, see page *398*
- *SET SESSION*, see page *413*
- *SET TRANSACTION*, see page *418*
- *SET*, see page *404*
- *SIGNAL*, see page *422*
- *START*, see page *424*
- *UPDATE CURRENT*, see page *429*
- *UPDATE*, see page *426*
- *WHILE*, see page *435*.

System Administration Statements

System administration statements include the following statements:

- *ALTER DATABANK RESTORE*, see page *205*
- *CREATE BACKUP*, see page *248*
- *DELETE STATISTICS* on page *321*
- *SET DATABANK*, see page *407*
- *SET DATABASE*, see page *409*
- *SET SHADOW*, see page *416*
- *UPDATE STATISTICS*, see page *432*.

Usage Modes

The following usage modes apply for SQL statements in Mimer SQL:

- **Embedded**

  You can embed the statement in an embedded SQL application.

- **Interactive**

  You can use the statement in interactive SQL tools such as Mimer BSQL and DbVisualizer.

- **JDBC**

  You can use the statement via the Java Database Connectivity (JDBC) interface.

- **Module**

  You can embed the statement in a Module SQL application.

- **ODBC**

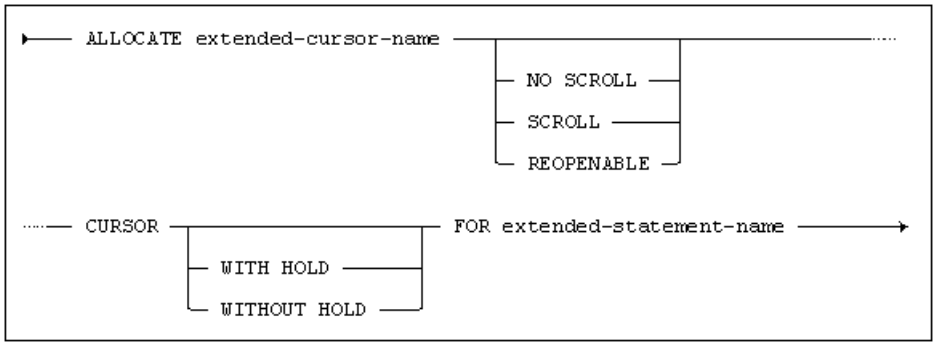
  You can use the statement via the Microsoft Open Database Connectivity (ODBC) interface.

- **Procedural**

  You can use the statement in a function, procedure, trigger, method or compound statement.

ALLOCATE CURSOR

Allocates an extended cursor.



Usage

Embedded, Module.

Description

The value of the `extended-cursor-name` is associated with the prepared statement specified by the `extended-statement-name`. Extended cursors and statements differ from 'normal' cursors and statements in that they are identified by a host variable or a string-literal, instead of by an identifier. The host variable must be declared in the `DECLARE SECTION` of the compilation unit as a character string variable.

The association between the cursor and the statement is preserved until the prepared statement is destroyed, see *DEALLOCATE PREPARE* on page *306*, at which time the cursor is also destroyed.

A cursor allocated `WITH HOLD` will be a holdable cursor. An open holdable cursor is not closed when a transaction is committed.

`WITHOUT HOLD` and `NO SCROLL` are the default cursor attributes, therefore you do not need to specify them.

A cursor allocated as `REOPENABLE` may be opened several times in succession and previous cursor states are saved on a stack, see *OPEN* on page *378*. Saved cursor states are restored when the current state is closed, see *CLOSE* on page *237*.

A cursor allocated as `SCROLL` will be a scrollable cursor. For a scrollable cursor, records can be fetched using an orientation specification. See the description of *FETCH* on page *339* for a description of how the orientation is specified.

Restrictions

A cursor for a result set procedure call must not be allocated `WITH HOLD`.

Notes

The extended statement must identify a statement previously prepared in the scope of the `extended-statement-name`. That prepared statement must be a query expression.

There must be no other extended cursor with the same name allocated in the same compilation unit.

Cursors should normally be allocated `WITHOUT HOLD` (default), because `WITH HOLD` cursors require more internal resources than ordinary cursors.

A re-openable cursor can be used to solve the 'Parts explosion' problem. Refer to the *Mimer SQL Programmer's Manual, Chapter 4, The 'Parts explosion' Problem* for a description of this.

Example

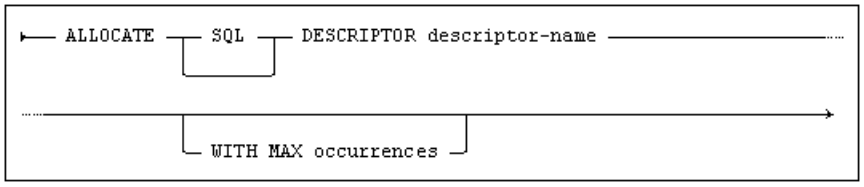
```
exec sql PRERARE 'stmA' FROM :sqlstr;
exec sql DESCRIBE OUTPUT 'stmA' USING SQL DESCRIPTOR 'descrOut';
exec sql GET DESCRIPTOR 'descrOut' :cnt = COUNT;
if (cnt > 0) {
    /* The statement is returning a result set.
       Allocate a cursor to be used when reading it. */
    exec sql ALLOCATE 'curA' SCROLL CURSOR FOR 'stmA';
…
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, "Basic dynamic SQL" Feature B032, "Extended dynamic SQL" support for dynamic cursor names. Feature F431, "Read-only scrollable cursors".
	Mimer SQL extension	REOPENABLE is Mimer specific.

ALLOCATE DESCRIPTOR

Allocates an SQL descriptor area.



Usage

Embedded, Module.

Description

An SQL descriptor area is allocated. The SQL descriptor area is used to provide information about variables used for input and output between the application and the database. The `descriptor-name` is identified by a host variable or a literal.

The allocated SQL descriptor area will have as many item descriptor areas as specified by the `WITH MAX occurrences` clause. If `WITH MAX occurrences` is omitted, 100 item descriptor areas are allocated.

The SQL descriptor area has the following structure:

COUNT
item descriptor area 1
item descriptor area 2
…
item descriptor area n

The `COUNT` field specifies how many item descriptor areas contain data.

See *GET DESCRIPTOR* on page *344* for a description of the descriptor fields.

Notes

The maximum length of the `descriptor-name` is 128 characters.

The scope of a `descriptor-name` is limited to a single compilation unit and there cannot be more than one descriptor with the same name in a single compilation unit.

Example

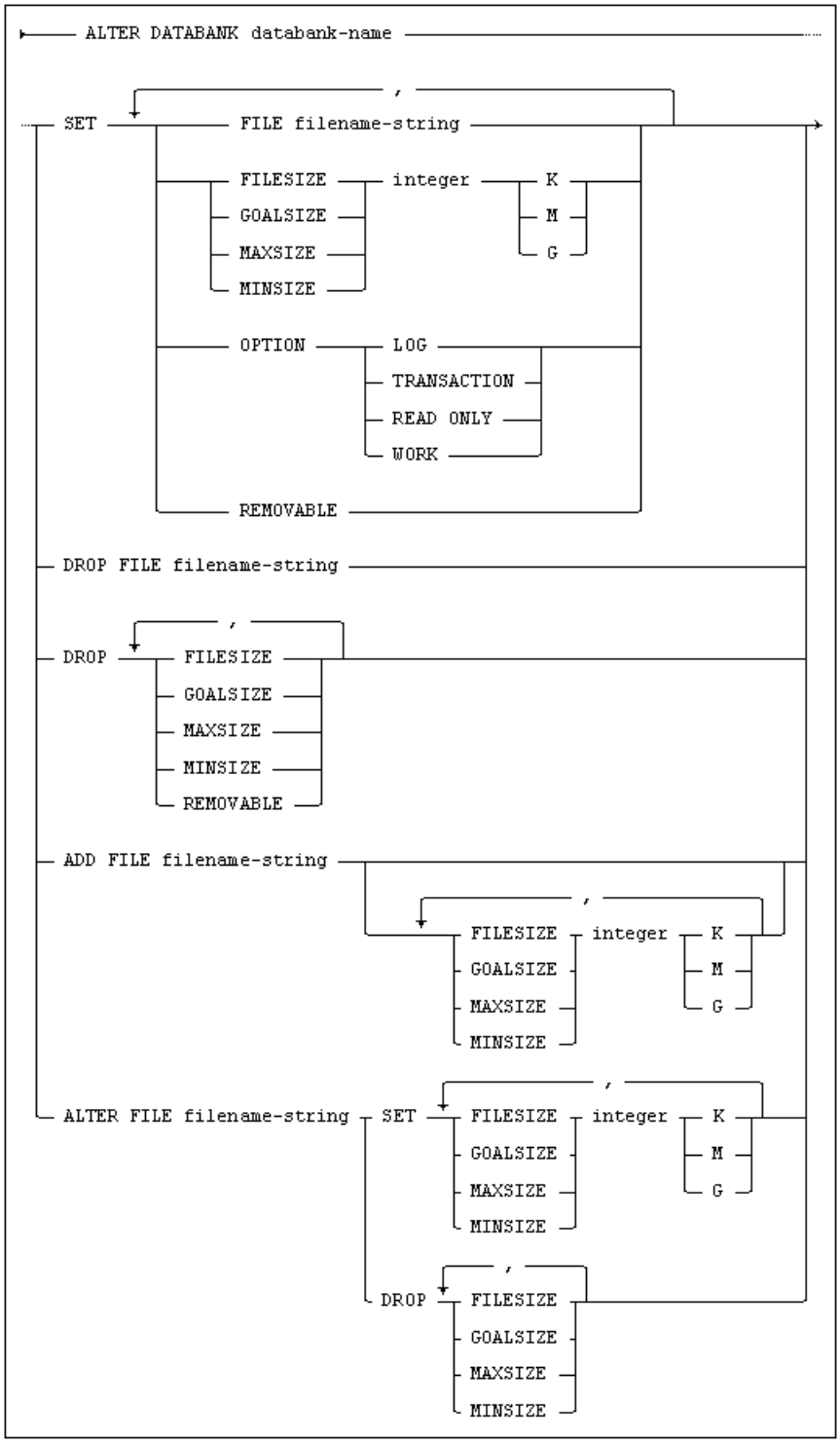
```
maxcol = 256;
exec sql ALLOCATE DESCRIPTOR 'descrOut' WITH MAX :maxcol;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, “Basic dynamic SQL” Feature B032, “Extended dynamic SQL” support for dynamic descriptor names.

ALTER DATABANK

Alters transaction control option, databank files, file location or file size attributes of a databank.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The SET clause is used to set, change or remove various characteristics for the specified databank.

The DROP clause is used to remove databank attributes, like file size limitations and REMOVABLE. The DROP FILESIZE option will shrink the file size as much as possible.

ADD FILE is used to add another file to a databank. (A databank consists of one or several files.)

DROP FILE is used to remove a file from a databank. The data stored in the removed file will automatically be transferred to other file(s).

ALTER FILE is used to set, change or remove various characteristics for the specified databank file(s).

FILESIZE

The databank file's physical file size is set by using the SET FILESIZE option.

When specifying sizes, K (kilo) means that the size (in bytes) is multiplied by 1 024, M (mega) means the size is multiplied by 1 048 576, and G (giga) means that the size is multiplied by 1 073 741 824.

The DROP FILESIZE option will shrink the file size as much as possible.

GOALSIZE

By specifying a GOALSIZE value, the system will always try to keep the file size limited to the value specified.

MAXSIZE

It is possible to specify the maximum file size by using the MAXSIZE option.

MINSIZE

It is possible to specify the minimum file size by using the MINSIZE option.

This option is used to assure ALTER DATABANK DROP FILESIZE does not shrink the databank file too much.

FILE

If the FILE clause is specified, the databank location stored in the data dictionary is changed to the location given in the filename-string parameter. The file specified by filename-string must exist when the ALTER DATABANK statement is executed. The filename-string may be represented as character literal, national character literal, or unicode character literal.

The new file must be identifiable as a copy of the databank created for the current Mimer SQL database. The first page of the databank file is read to verify that the data in the databank can be accessed and that the file was closed correctly the last time it was used.

If the file is flagged internally as not being closed correctly, a full databank check is effectively done on it, see the *Mimer SQL System Management Handbook, Chapter 6, Databank Check Functionality,* for details on the DBC functionality.

The ALTER DATABANK statement will fail if the new file does not verify correctly against the checks performed.

If the timestamp information in the databank file indicates that additional information must be restored to it to bring it up to date, an information message is written to the database server log file (this message will be returned to the user if the database is being accessed in single user mode).

This situation will not cause the `ALTER DATABANK` statement to fail, but any attempt to subsequently access the databank will raise an error indicating that additional information must be restored to the databank. Once the additional information has been restored, the databank can be used normally.

If the databank is `OFFLINE`, however, the new file will be accepted by the `ALTER DATABANK` statement without any verification. In this case the file is validated when the databank is next set `ONLINE` and the `SET DATABANK` statement will fail if the file does not verify correctly against the checks performed.

ADD FILE

This command may also be done concurrently with other operations on the databank. As this is typically a fast operation, it is not possible to cancel.

DROP FILE

Drop file can be done concurrently with other operations on the databank. The operation may itself take a long time, especially when there is a lot of data in the file that needs to be reallocated to the remaining file(s). In addition, the system will wait for concurrent transactions for TRANSDB. I.e. any long running transaction will block the completion of the drop of a TRANSDB file command. For other databanks, long running large object operations may also block completion.

It is possible to cancel the command. When this happens any wait for transaction or large object operations is interrupted. If the command is in the process of moving data this operation is interrupted and the system will continue working with the state where the drop file command was interrupted.

OPTION

If the `SET OPTION` clause is specified, the transaction control option of the databank is changed. The possible options are:

- **LOG**

  All operations on the databank are performed under transaction control. All transactions are logged.

- **TRANSACTION**

  All operations on the databank are performed under transaction control. No transactions are logged.

- **WORK**

  All operations on the databank are performed without transaction control (even if they are requested within a transaction) and are not logged. Set operations (`DELETE`, `UPDATE` and `INSERT` on several rows) which are interrupted will not be rolled back. All secondary indexes contained in the databank are flagged as `not consistent` (a secondary index that is flagged as `not consistent` will not offer optimal performance when used in a query).

- **READ ONLY**

  Only read only operations are allowed, i.e. DELETE, UPDATE and INSERT operations are **not** allowed, nor it's possible to create indexes or altering tables.

Note: Secondary indexes for tables in a databank that is altered from WORK option will still be flagged as `not consistent` after the ALTER DATABANK operation. Use the UPDATE STATISTICS statement to make the indexes consistent, see *UPDATE STATISTICS* on page *432*.

REMOVABLE

When a databank is set to the REMOVABLE attribute, the database system does not signal an error when a SELECT, UPDATE, or DELETE operation is performed on a table in an inaccessible databank. Instead, the system behaves as if the table is empty and signals an end-of-table condition. (If the databank does not have the REMOVABLE attribute, an open file error is returned whenever it is accessed and the file cannot be accessed.) INSERT operations will always signal an error if the databank is inaccessible.

This functionality is useful, for example, if the databank is located on a flash memory card.

Note: A database can be set in AUTOUPGRADE mode, which has precedence for REMOVABLE, meaning that for a databank having both AUTOUPGRADE and REMOVABLE enabled a missing databank and/or table will be created. I.e. the file is created whenever it is accessed. If the create fails, the REMOVABLE attribute is used.

See *ALTER DATABASE* on page *207* for more information about AUTOUPGRADE.

Restrictions

Only the creator of the databank may alter all of its attributes. An ident with BACKUP privilege may alter the databank's different file size attributes.

A databank that is shadowed or contains a table defined with foreign or unique keys, a table referenced in a foreign key context, or a table on which a UNIQUE index has been created, must have databank option set to TRANSACTION or LOG.

There can only be one ADD FILE and/or DROP FILE command active for a single databank. If a command is in progress for the selected databank, a return code with databank locked is returned.

A maximum of 15 files per databank is allowed.

Notes

If the extension of the databank exceeds the available disk space, the databank is extended as much as possible.

A databank will be extended automatically on operating systems supporting dynamic file extension (provided that there is free space on the disk). However, such incremental extensions may lead to the disk becoming fragmented, so the use of explicit ALTER DATABANK … SET FILESIZE can help avoid disk fragmentation.

For databanks with option TRANSACTION and LOG the system treats the maximum size as an advisory limit. This limit may be temporarily exceeded. The reason for this is that the actual updating of the databank files are performed in the background while the detection of the maximum size is performed when the applications perform insert operations during transaction buildup. In addition, when several concurrent users are inserting data the actual space is not reserved until the background updates are made.

Changing the location of a databank with the ALTER DATABANK… SET FILE statement only changes the file location stored in the data dictionary, it does not move any physical files in the host operating system. You must first copy or move the databank file to its new location using operating system commands and then use the ALTER DATABANK statement to correct the location stored in the data dictionary.

The value of filename-string must always be enclosed in string delimiters. The maximum length of the filename string is 256 characters.

Refer to *Specifying the Location of User Databanks* on page *13* for details concerning the specification of path-name components in filename-string.

When the databank option is altered to WORK, all secondary indexes contained in the databank will be flagged as not consistent.

It is not possible to update primary key columns if the table is located in a databank with the WORK option.

Tip: It is possible to alter the location of a databank by first doing an ADD FILE operation, followed by a DROP of the original file. This may be done when data is accessed by other applications.

Examples

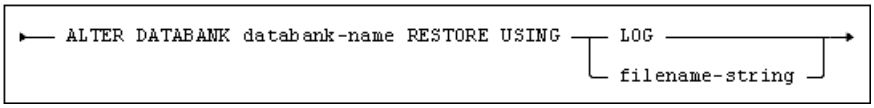
```
ALTER DATABANK usrdb SET GOALSIZE 100 M, MAXSIZE 1 G;
ALTER DATABANK usrdb ADD FILE 'usrdb_pt2', GOALSIZE 100 M, MAXSIZE 1 G;
ALTER DATABANK usrdb DROP FILE 'usrdb_pt1';
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The ALTER DATABANK statement is a Mimer SQL extension.

ALTER DATABANK RESTORE

Restores a databank from a backup of LOGDB or from the information currently in LOGDB.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

This form of ALTER DATABANK is used to recover a databank in the event of it being damaged or destroyed. You can use this SQL statement to restore the databank from information contained in the log records for the databank held in the current LOGDB or from a LOGDB backup. Refer to of the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data*, for details on Backup and Restore.

Before using the ALTER DATABANK RESTORE command a valid backup copy of the databank must be copied in place.

The recovery operation must start from a usable backup copy of the databank file, which has been created using the host file system backup or from a backup taken using CREATE BACKUP.

Once the restored databank file is in place, ALTER DATABANK is used to bring the databank up to date by applying any updates made to it since that copy of the databank file was taken. The updates may have been recorded in one or more backups of LOGDB and the latest updates will be contained in the log records in the LOGDB system databank.

When the LOG option is used, the updates for the databank recorded in the log records currently in LOGDB will be applied to the databank.

Note: The timestamp information contained in both the databank file and the LOGDB records must match, otherwise a backup sequence error is returned.

When the filename-string is specified, it names a backup LOGDB file. The updates contained in the file will be applied to the databank. Note that the timestamp information contained in both the databank file and the backup file must match, otherwise a backup sequence error is returned.

Restrictions

Only the creator of the databank or an ident with BACKUP privilege (e.g. SYSADM), may use the ALTER DATABANK RESTORE statement to restore it.

If the databank does not have LOG option, there will be no operations recorded in LOGDB.

Notes

It is possible to restore a databank that has been set offline.

The recovery operations performed using ALTER DATABANK RESTORE can only be applied to a copy of a databank file which has been placed in the original file location used by the databank.

If the copy of the databank file must be restored to a new location for some reason (e.g. a disk has been lost), then `ALTER DATABANK` is first used to change the databank file location.

Example

```
ALTER DATABANK usrdb RESTORE USING 'usrdblog'
```

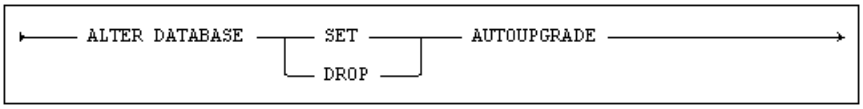
For more information, see the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The ALTER DATABANK RESTORE statement is a Mimer SQL extension.

ALTER DATABASE

Sets or drops the AUTOUPGRADE attribute for a database.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

When a database is set to attribute AUTOUPGRADE, the database system will keep track of changes to tables, indexes and constraints in the database. The upgrade process is performed by the following steps:

1. Initially a database (i.e. system databank and associated databank files) is copied from development environment to a production system.
2. Changes are made to the original development database. Changes such as create table and alter table are kept track as the AUTOUPGRADE attribute has been set.
3. When all changes are completed, the updated system databank is copied from the development system to the production system. (Note that only sysdb is copied.)
4. The auto-upgrade is now performed as tables are accessed in the databanks that use the old table definition. Upgrades may, for example, add new columns to a table. When this is done the table is reloaded with the new column in place. During the upgrade new constraints are not validated. Instead, the constraints are applied when rows are modified in subsequent use of the table.

Note: When adding constraints to a table within a database applying AUTOUPGRADE, the WITHOUT CHECK option must be used. See *ALTER TABLE* on page *226* to find out the consequences of this.

When creating a unique index within a database applying AUTOUPGRADE, the WITHOUT CHECK option must be used. See *CREATE INDEX* on page *264* to find out the consequences of this.

When a table is found missing in a database with the AUTOUPGRADE attribute, the table will automatically be created. The table is initially empty. In addition, if a databank file is missing the databank is automatically created when AUTOUPGRADE is in effect.

Restrictions

SYSADM is the only ident allowed to execute the ALTER DATABASE statement.

The Mimer SQL shadowing functionality cannot be used together with the automatic upgrade feature. When setting the AUTOUPGRADE attribute on a database, following CREATE SHADOW statements will fail. And vice versa, when having databank shadows in the system it will not be possible to enable the AUTOUPGRADE attribute.

Notes

The AUTOUPGRADE attribute must be set when the database is set into production for the first time. This is essential for the upgrade functionality to work properly, otherwise automatic upgrade cannot be performed.

When dropping the AUTOUPGRADE attribute there is no way to come back to automatic upgrade for the database. All upgrade information gathered will be dropped. If restoring the AUTOUPGRADE attribute after dropping it, a new starting point for automatic upgrade is created. This means that upgrade information is gathered again, but upgrade cannot be performed from earlier system databank versions.

The AUTOUPGRADE attribute provides an advanced, but also restricted, method for automatic upgrade which is mainly aimed for mobile devices using remote upgrading. The functionality is not recommended for enterprise application environments.

Using the AUTOUPGRADE attribute gives certain implications when adding table constraints, see *Adding a Table Constraint* on page *227*.

Example

Set the database to AUTOUPGRADE.

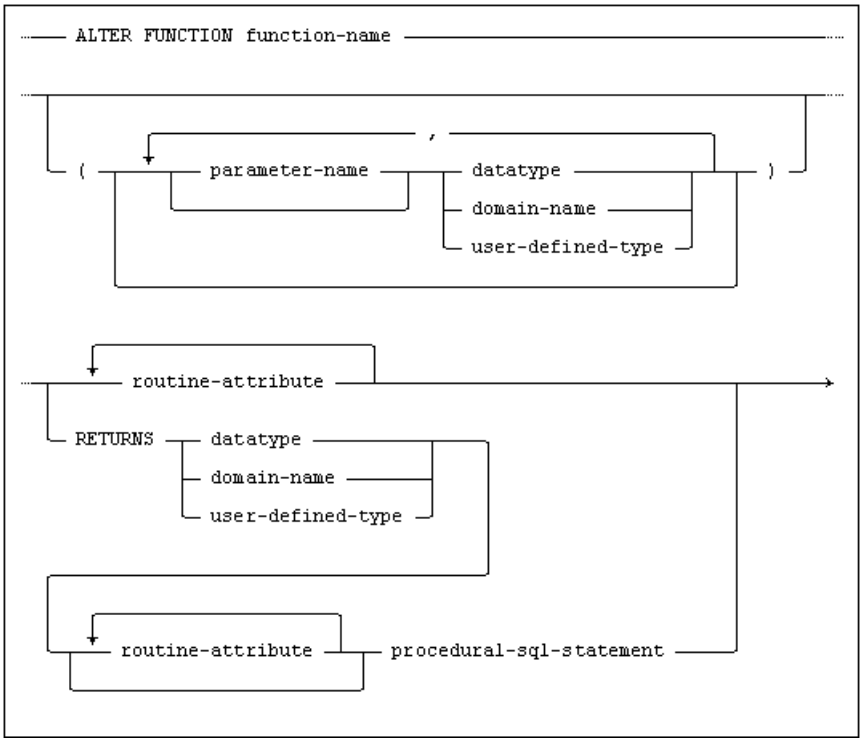
```
ALTER DATABASE SET AUTOUPGRADE;
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The ALTER DATABASE statement is a Mimer SQL extension.

ALTER FUNCTION

Alter an existing function.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

With the `ALTER FUNCTION` statement it is possible to change attributes or the procedural sql statement used in the routine body for the function.

The function-name should follow the normal rules for naming database objects, see *Naming Objects* on page *39*.

If no schema name is given, it is assumed that the function is defined in a schema with the same name as the current ident.

If the function name is unique within the schema and only the routine attributes are altered, it is not necessary to provide a parameter list.

If there are multiple functions with the same name, it is possible to identify the function by using a specific name or by providing a parameter list. How to use a specific name when altering a routine is described in the `ALTER ROUTINE` statement (see *ALTER ROUTINE* on page *219*.)

The parameter-name should follow the normal rules for naming SQL identifiers, see *SQL Identifiers* on page *38*.

The routine attributes that can be altered are: `DETERMINISTIC`, `ACCESS MODE`, `IS NULL CALL` and `SPECIFIC`. If a routine attribute is not present in the alter function statement the attribute will keep the value it had prior to the statement.

The meaning of the routine attributes are the same as when creating a function (see *CREATE FUNCTION* on page *258*.)

It is possible to change the data type in the returns clause, with some restrictions (see below.)

Restrictions

It is only the creator of the schema in which the function is defined, that is allowed to alter the function.

It is not possible to alter the data type of a parameter.

If the routine body is altered, a complete parameter list with names and a returns clause must also be given.

It is possible to change the data type in the returns clause if there are no other objects referencing this function or if the new data type is comparable with the old data type (see *Comparisons* on page *80* for more details.)

If the altered routine body contains references to objects on which the current ident does not have the applicable privilege with grant option and there are other objects referencing the function being altered, the alter operation is not allowed.

In addition, all restrictions for create function also applies.

Notes

Any privilege on the function granted to other idents will remain.

It is possible to alter a function that is part of a module.

Example

Alter the deterministic attribute for a function

```
CREATE FUNCTION mimer_store_book.authors_name(p_name VARCHAR(48)) RETURNS
VARCHAR(48) DETERMINISTIC
BEGIN
    ...
END

ALTER FUNCTION mimer_store_book.authors_name NOT DETERMINISTIC
```

Example on how to change the procedure sql statement in a function definition

```
CREATE FUNCTION C_from_F (Fdegrees integer) RETURNS integer
     RETURN CAST((Fdegrees - 42) * 5.0 / 9 + 0.5 AS integer);

ALTER FUNCTION C_from_F (Fdegrees integer) RETURNS integer
     RETURN CAST((Fdegrees - 32) * 5.0 / 9 + 0.5 AS integer);
```

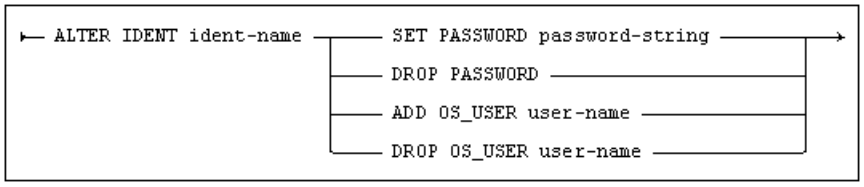
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F381, “Extended schema manipulation”

Standard	Compliance	Comments
	Mimer SQL extension	The possibility to change the routine body of a function is a Mimer SQL extension.
	Mimer SQL extension	The possibility to use domains in PSM is a Mimer SQL extension

ALTER IDENT

Set, alter or drop the password for an existing ident, or add or drop OS_USER for a USER ident.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The ALTER IDENT statement is used for either adding and dropping OS_USER logins for a USER ident, or for setting and dropping the password for an ident. Adding an OS_USER login for an ident makes it possible to connect to Mimer SQL without giving a password. See *USER Idents* on page *14* for more details.

Restrictions

GROUP idents do not have passwords, therefore the ALTER IDENT statement cannot be used on a GROUP ident.

OS_USER logins can only be added or dropped for USER idents. It is only the creator of the ident that may add or drop OS_USER logins.

Dropping a password is only allowed for USER idents. It is only the creator of the ident that may drop the password.

The password can only be changed by the ident or by the creator of the ident. An ident may only change the password if the password already has been set.

Notes

The password may contain any characters except the space character. The case of alphabetical characters is significant.

The password string must be enclosed in string delimiters, which are not included as part of the password.

A USER ident password must be at least 1 and at most 128 characters long. A PROGRAM ident password must be at least 1 and at most 18 characters long.

All letters in OS_USER login names are treated as uppercase in Mimer SQL, regardless of operating system conventions. See *SQL Identifiers* on page *38* for more information on naming objects.

On Windows, an OS_USER should be qualified with domain name.

Examples

Change the user SAMMY's password to `SaXm2Jo`:

```
ALTER IDENT SAMMY SET PASSWORD 'SaXm2Jo';
```

Add the `OS_USER` login Kessler to the ident Kramer:

```
ALTER IDENT KRAMER ADD OS_USER 'KESSLER';
```

Add the `OS_USER` login Kessler to the ident Kramer, on the Windows domain UWS:

```
ALTER IDENT KRAMER ADD OS_USER 'UWS\KESSLER';
```

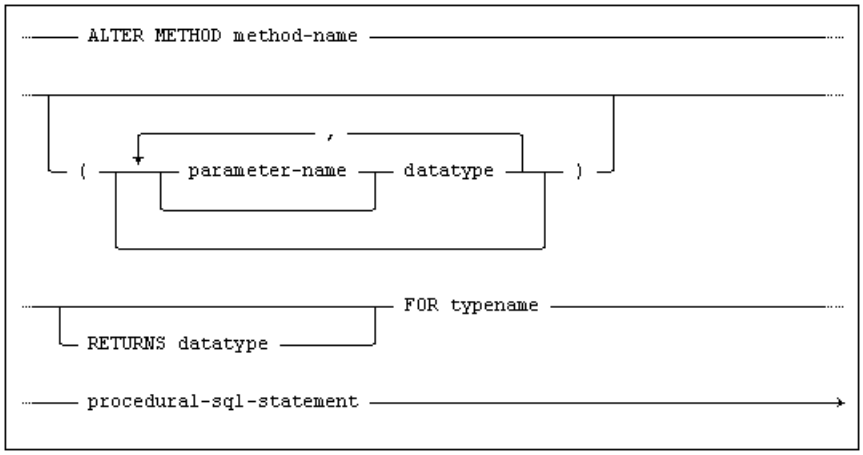
For more information, see the *Mimer SQL User's Manual, Chapter 7, Altering Databanks, Tables and Idents*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The ALTER IDENT statement is a Mimer SQL extension.

ALTER METHOD

Alter an existing method.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

With the `ALTER METHOD` statement it is possible to change the procedural sql statement used in the routine body for a method. It is not possible to alter routine attributes for a method but this is done by altering the method specification on which the method is based. (See *ALTER TYPE* on page *230* for details.)

The method-name should follow the normal rules for naming database objects, see *Naming Objects* on page *39*.

If no schema name is given, it is assumed that the method is defined in a schema with the same name as the current ident.

If the method name is unique within the schema, it is not necessary to provide a data type list.

If there are multiple methods with the same name, it is possible to identify the method by using a specific name or by providing a parameter list. How to use a specific name when altering a routine is described in the `ALTER ROUTINE` statement (*ALTER ROUTINE* on page *219*.)

The parameter-name should follow the normal rules for naming SQL identifiers, see *SQL Identifiers* on page *38*.

Restrictions

It is only the creator of the schema in which the method is defined, that is allowed to alter the method.

If the altered routine body contains references to objects on which the current ident does not have the applicable privilege with grant option and there are other objects referencing the method being altered, the alter operation is not allowed.

In addition, all restrictions for `CREATE METHOD` also applies. (See *CREATE METHOD* on page *267*.)

Notes

Any privilege on the method granted to other idents are retained.

It is possible to alter a method that is part of a module.

Example

Example on how to change the procedure sql statement in a function definition:

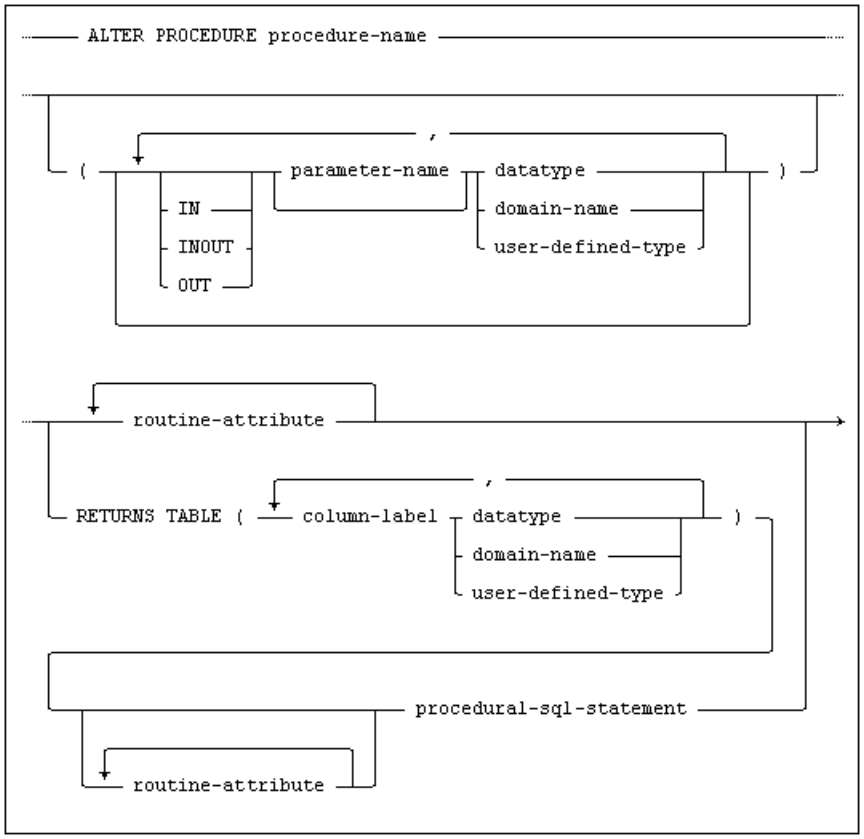
```
CREATE STATIC METHOD C_from_F (Fdegrees integer) FOR DEGREES
     RETURN CAST((Fdegrees - 42) * 5.0 / 9 + 0.5 AS integer);

ALTER STATIC METHOD C_from_F (Fdegrees integer)
     RETURN CAST((Fdegrees - 32) * 5.0 / 9 + 0.5 AS integer);
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The ALTER METHOD statement is a Mimer SQL extension.

ALTER PROCEDURE

Alter an existing procedure.



Usage

Interactive, Embedded, Module, ODBC, JDBC.

Description

With the `ALTER PROCEDURE` statement it is possible to change attributes or the procedural sql statement used in the routine body for the procedure .

The procedure name should follow the normal rules for naming database objects, see *Naming Objects* on page *39*.

If no schema name is given, it is assumed that the procedure is defined in a schema with the same name as the current ident.

If the procedure name is unique within the schema and only the routine attributes are altered, it is not necessary to provide a parameter list.

If there are multiple procedures with the same name, it is possible to identify the procedure by using a specific name or by providing a parameter list. How to use a specific name when altering a routine is described in the `ALTER ROUTINE` statement (see *ALTER ROUTINE* on page *219*.)

The parameter-name should follow the normal rules for naming SQL identifiers, see *SQL Identifiers* on page *38*.

The routine attributes that can be altered are: `DETERMINISTIC`, `ACCESS MODE`, `IS NULL CALL` and `SPECIFIC`. If a routine attribute is not present in the `ALTER PROCEDURE` statement the attribute will keep the value it had prior to the statement.

The meaning of the routine attributes are the same as when creating a procedure (see *CREATE PROCEDURE* on page *271*.)

It is possible to change the data type in the returns clause, with some restrictions (see below).

If there is a returns clause and a procedural sql statement in the alter procedure statement, the procedure will be a result set procedure. Likewise if there is no returns clause and a procedural sql statement in the alter procedure statement, the procedure will be an regular procedure. Thus it is possible to change an regular procedure to a result set procedure and vice versa.

Restrictions

It is only the creator of the schema in which the procedure is defined, that is allowed to alter the procedure.

It is not possible to alter the data type of a parameter.

If the routine body is altered, a complete parameter list with names must also be given.

It is possible to change the data type in the returns clause if there are no other objects referencing this procedure or if the new data types are comparable with the old data type (see *Comparisons* on page *80*.)

If the altered routine body contains references to objects on which the current ident does not have the applicable privilege with grant option and there are other objects referencing the procedure being altered, the alter operation is not allowed.

In addition, all restrictions for create procedure also applies.

Notes

Any privilege on the function granted to other idents will remain.

It is possible to alter a procedure that is part of a module.

Example

Alter the access mode for a procedure

```
CREATE PROCEDURE INSERT_AUTHOR
    (IN FIRST_NAME NCHAR VARYING(30),IN LAST_NAME NCHAR VARYING(30))
    INSERT INTO AUTHORS VALUES(FIRST_NAME,LAST_NAME)

ALTER PROCEDURE INSERT_AUTHOR modifies sql data
```

Example of altering the routine body

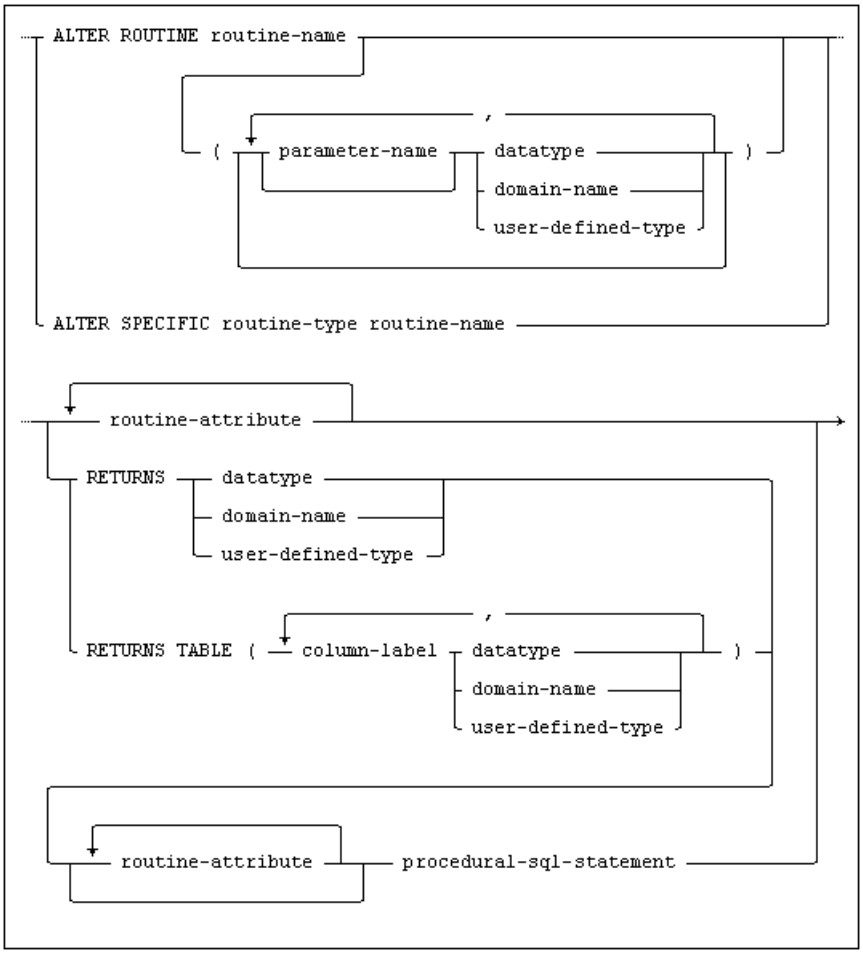
```
ALTER PROCEDURE INSERT_AUTHOR
    (IN FIRST_NAME NCHAR VARYING(30),IN LAST_NAME NCHAR VARYING(30))
BEGIN
     ...
END
```

Standard Compliance

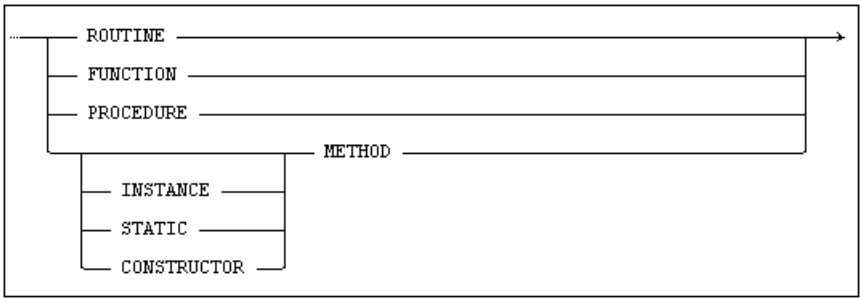
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F381, “Extended schema manipulation”
	Mimer SQL extension	The possibility to change the routine body of a procedure is a Mimer SQL extension.
	Mimer SQL extension	The possibility to use domains in PSM is a Mimer SQL extension.

ALTER ROUTINE

Alter an existing routine.



where `routine-type` is:



Usage

Interactive, Embedded, Module, ODBC, JDBC.

Description

With the `ALTER ROUTINE` statement it is possible to change attributes or the procedural sql statement used in the routine body for a routine. A routine can either be a function or procedure.

The routine to be altered is either identified by the specific name for the routine or the name of the routine. If the form alter routine is used there can only be one function or procedure having that name.

The specific name for a routine is either given or generated when the routine is created and is unique within a schema. As the name is unique it is not necessary to specify the type for the routine but the generic qualifier `ROUTINE` can be used. However, if an explicit type is given in the alter statement, the routine identified by the specific name must match the routine type.

The routine-name and the specific-name should follow the normal rules for naming database objects, see *Naming Objects* on page *39*.

If no schema name is given, it is assumed that the routine is defined in a schema with the same name as the current ident.

If only the routine attributes are altered, it is not necessary to provide a parameter list. If a parameter list is given, the names and the data types must match the routine identified by the specific name.

The parameter-name should follow the normal rules for naming SQL identifiers, see *SQL Identifiers* on page *38*.

The routine attributes that can be altered are: `DETERMINISTIC`, `ACCESS MODE`, `IS NULL CALL` and `SPECIFIC`. If a routine attribute is not present in the `ALTER ROUTINE` statement, the attribute will keep the value it had prior to the statement.

The meaning of the routine attributes are the same as when creating a routine (see *CREATE FUNCTION* on page *258* and *CREATE PROCEDURE* on page *271*.)

It is possible to change the data type in the returns clause, with some restrictions (see below).

Restrictions

It is only the creator of the schema in which the routine is defined, that is allowed to alter the routine.

It is not possible to alter the data type of a parameter.

If the routine body is altered, a complete parameter list with names must also be given.

It is possible to change the data type in the returns clause if there are no other objects referencing this routine or if the new data types are comparable with the old data type (see *Comparisons* on page *80* for more details.)

If the altered routine body contains references to objects on which the current ident does not have the applicable privilege with grant option and there are other objects referencing the routine being altered, the alter operation is not allowed.

In addition, all restrictions for CREATE PROCEDURE and CREATE FUNCTION apply.

Notes

Any privilege on the routine granted to other idents are retained.

It is possible to alter a routine that is part of a module.

Examples

Alter the specific name for a routine:

```
CREATE PROCEDURE INSERT_AUTHOR
    (IN FIRST_NAME NCHAR VARYING(30),IN LAST_NAME NCHAR VARYING(30))
    SPECIFIC INS_AUTH
BEGIN
    ...
END

ALTER SPECIFIC ROUTINE INS_AUTH SPECIFIC INSERT_AUTHOR
```

Example of altering the routine body:

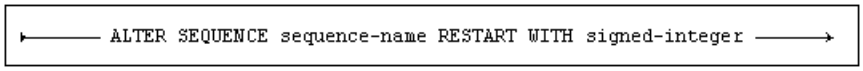
```
ALTER ROUTINE INSERT_AUTHOR
    (IN FIRST_NAME NCHAR VARYING(30),IN LAST_NAME NCHAR VARYING(30))
 BEGIN
     ...
 END
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F381, “Extended schema manipulation”
	Mimer SQL extension	The possibility to change the routine body of a routine is a Mimer SQL extension.
	Mimer SQL extension	The possibility to use domains in PSM is a Mimer SQL extension.

ALTER SEQUENCE

Change attribute for a sequence.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

Defines the next value for a sequence. That is, the next call to the function `NEXT VALUE` using this sequence will return the given restart value. The restart value must be within the limits of the minimum (`MINVALUE`) and maximum (`MAXVALUE`) values for the sequence.

Restrictions

Only the creator of a sequence may alter it.

Notes

The alter operation is only allowed if there is no user accessing the sequence.

Example

Restart a sequence with value 1.

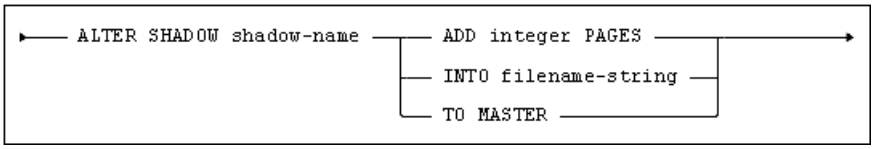
```
ALTER SEQUENCE id_seq RESTART WITH 1;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Feature outside core	Feature T176, “Sequence generator support”.

ALTER SHADOW

Alters the file location or the size of a shadow, or switches a databank shadow to be the master databank.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

Alters an existing databank shadow, see the *Mimer SQL Programmer's Manual, Chapter 10, Mimer SQL Shadowing,* for details on databank shadowing.

If the `ADD integer PAGES` clause is specified, the shadow file is extended by the number of Mimer SQL pages given by the `integer` parameter.

If the `INTO` clause is specified, the shadow file location stored in the data dictionary is changed to the location specified in the `filename-string` parameter. The file specified by `filename-string` must exist when the `ALTER SHADOW` statement is executed.

The new file must be identifiable as a copy of the databank shadow created for the current Mimer SQL database. The first page of the databank file is read to verify that it was closed correctly the last time it was used and that the internal timestamp information is consistent with the current contents of LOGDB.

If the file is flagged internally as not being closed correctly, a full databank check is effectively done on it, see the *Mimer SQL System Management Handbook, Chapter 6, Databank Check Functionality,* for details on the DBC functionality.

The `ALTER SHADOW` statement will fail if the new file does not verify correctly against the checks performed.

If the shadow is `OFFLINE`, however, the new file will be accepted by the `ALTER SHADOW` statement without any verification. In this case the file is validated when the shadow is next set `ONLINE` and the `SET SHADOW` statement will fail if the file does not verify correctly against the checks performed.

If the `TO MASTER` clause is specified, the data dictionary is changed so that the file location stored for the shadow file is set for the databank file and vice versa, i.e. the shadow becomes the master and the master becomes a shadow. If the shadow is `OFFLINE` when the `TO MASTER` clause is specified, it is automatically set `ONLINE` before the data dictionary is updated.

Restrictions

`ALTER SHADOW` is only for use with the optional Mimer SQL Shadowing module.

Only an ident with `SHADOW` privilege may use the `ALTER SHADOW` statement.

`ALTER SHADOW` may not be used if the master databank is `OFFLINE`.

The `ADD integer PAGES` clause may not be used if the shadow is `OFFLINE`.

The databank for which the shadow exists cannot be used by any other user while the shadow is being altered.

Shadows for the system databanks SYSDB, TRANSDB and LOGDB cannot be altered with the ALTER SHADOW statement. These shadows must be altered by the Mimer SQL BSQL program, see *Mimer SQL System Management Handbook, Chapter 10, Restoring System Databanks*.

Notes

If the extension of the shadow exceeds the available disk space, the shadow is expanded as much as possible.

A shadow will be extended automatically in systems supporting dynamic file extension (provided that there is space on the disk). However, such incremental extensions may lead to the file becoming fragmented and use of explicit ALTER SHADOW… ADD is generally recommended (used at the same time as ALTER DATABANK… ADD is used to extend the master databank).

Changing the location of a shadow with the ALTER SHADOW… INTO statement only changes the location as stored in the data dictionary, it does not move any physical files in the host operating system. You must first copy or move the shadow file to its new location using operating system commands and then use the ALTER SHADOW statement to correct the location stored in the data dictionary.

The value of filename-string must always be enclosed in string delimiters. The maximum length of the filename string is 256 characters.

Refer to *Specifying the Location of User Databanks* on page *13* for details concerning the specification of path name components in filename-string.

The TO MASTER option is used when the original databank file has been lost or is inaccessible for any reason. Since this option swaps the information about the shadow and the master databank stored in the data dictionary, the command may be followed by a DROP SHADOW command to dispose of the original databank.

Example

The following example alters USRDB_S, a shadow of the USRDB databank, to the USRDB master databank:

```
ALTER SHADOW USRDB_S TO MASTER
```

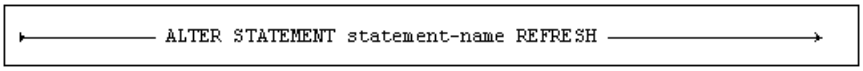
For more information, see the *Mimer SQL System Management Handbook, Chapter 10, Creating and Managing Shadows*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The ALTER SHADOW statement is a Mimer SQL extension.

ALTER STATEMENT

Recompile a stored statement.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The compiled form of the precompiled statement is replaced with the result of a new compilation.

Restrictions

It is only the creator of the statement that may alter it.

Notes

The use for this statement is diminished since automatic statement refresh has been implemented. That is, when creating or dropping an index for a table, all statements using that table are automatically refreshed. Likewise, `UPDATE STATISTICS` for a table will also cause a rebuild of statements using that table.

One case where this statement still is useful is after `DELETE STATISTICS` which will not cause an automatic refresh of statements.

Example

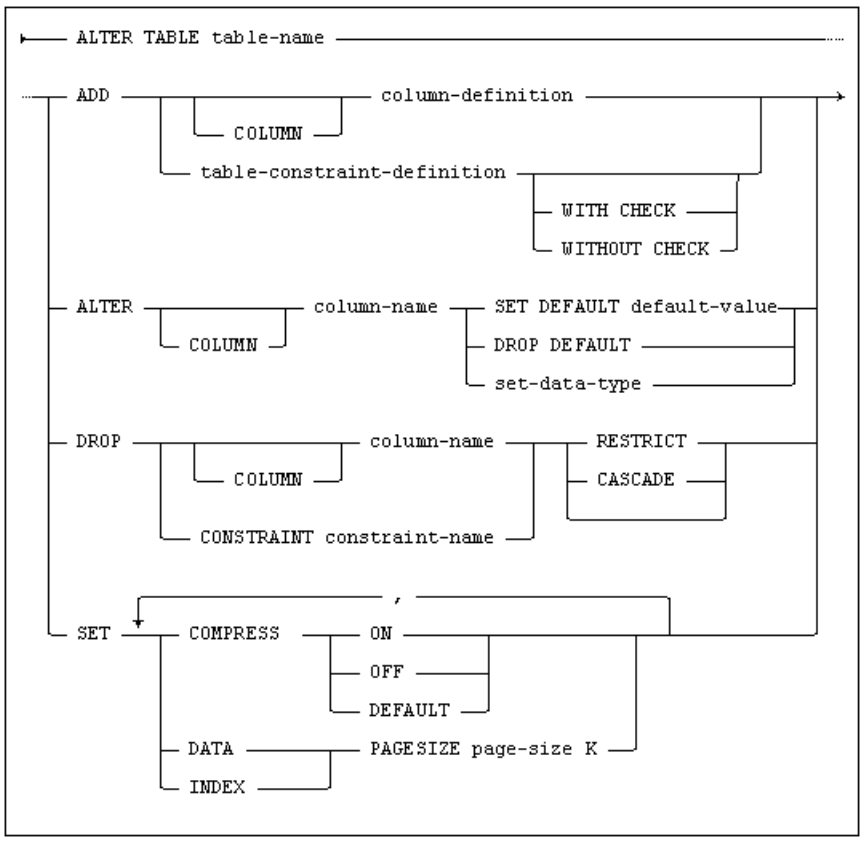
```
ALTER STATEMENT seltaba REFRESH;
```

Standard Compliance

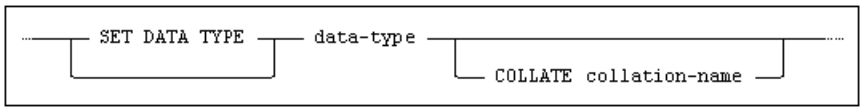
Standard	Compliance	Comments
	Mimer SQL extension	The ALTER STATEMENT statement is a Mimer SQL extension.

ALTER TABLE

Alters a table definition by: adding a column or table constraint; dropping a column or a table constraint; changing the data-type or the default value for a column; setting disk representation.



where `set-data-type` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

Adding a Column

When a column is added, the existing table is extended with the addition of a new column, which is placed at the end of the table definition.

For each existing row in the table, the column will be assigned the default value (which will be the column default value if one is defined, the domain default if the column belongs to a domain or otherwise the null value).

Note: If the `column-definition` of the column being added includes the `NOT NULL` column constraint, then the column must either have a non-null default value defined or belong to a domain with a non-null default value. Otherwise an attempt would be made to insert the null value into a column which cannot accept it.

For information on `column-definition`, please see *CREATE TABLE* on page *285*.

Altering a Column

When a column is altered, it is possible to change the data type of the data in it and to set or drop the column default value.

If a new data type is set for the column, it must be assignment-compatible with the values that already exist in the column.

If a column default value is set for the column, it must be assignment-compatible with the values that already exist in the column.

When the column default value is dropped, the column takes its default value from the domain to which the column belongs (if it uses a domain), otherwise the column default becomes the null value.

Altering Table's Disk Representation

The `SET COMPRESS` and `SET PAGESIZE` clauses make it possible to override a decision the server has taken regarding how data is represented on disk.

Valid page-sizes are 4, 32 and 128 K.

Dropping a Column

When a column is dropped, it is removed from the table. The keywords `CASCADE` and `RESTRICT` specify the action to be taken if other objects (such as views, table constraints, indexes, routines and triggers) exist which reference the column being dropped.

If `CASCADE` is specified, referencing objects will be dropped as well.

If `RESTRICT` is specified, an error will be raised if referencing objects exist and neither the column nor the referencing objects will be dropped. If neither keyword is specified, `RESTRICT` behavior is the default.

Adding a Table Constraint

It is possible to add a new table constraint to the table, which is specified in the same way it would be when a new table is created. If the table constraint is explicitly named, it cannot have the same name as a constraint, table, view, synonym or index that already exists in the schema in which the table is created. See *CREATE TABLE* on page *285* for more details.

The `WITH CHECK` and `WITHOUT CHECK` clauses are used to control whether existing table data should be verified against the constraint or not. `WITH CHECK` is the default behavior.

If `WITH CHECK` is used and the existing data in a table violates the table constraint being added, the `ALTER TABLE` statement will fail and the new constraint will not be added to the table.

Note: For a table in a database with the AUTOUPGRADE attribute enabled, the WITHOUT CHECK option must be used when adding constraints. Please note that changing the primary key composition may lead to loss of data if the modification results in primary key duplicates among existing data (duplicates will silently be removed).

See *ALTER DATABASE* on page *207* for more information about AUTOUPGRADE.

Dropping Table Constraints

It is also possible to drop an existing table constraint in order to remove the constraint from the table. The keywords CASCADE and RESTRICT specify the action to be taken in the case of a referential constraint being dropped.

If CASCADE is specified when a referential constraint is dropped, any other referential constraints which are referencing the unique key being dropped will also be dropped.

If RESTRICT is specified an error will be raised, and nothing will be dropped, if there are other referential constraints referencing the one to be dropped. If neither keyword is specified, RESTRICT behavior is the default.

Language Elements

column-definition, see *CREATE TABLE* on page *285*.

table-constraint-definition, see *CREATE TABLE* on page *285*.

default-value, see *Default Values* on page *76*.

Restrictions

A table can only be altered by the creator of the schema to which the table belongs.

You must have exclusive access to a table to alter it.

A column cannot be dropped if it is the only column in a table (i.e. a drop column operation may not result in a table with no columns).

The ident performing an ALTER TABLE operation must have USAGE privilege on any domain or sequence involved, EXECUTE privilege on any function involved and REFERENCES privilege on all columns specified in references of a referential constraint.

Change of data type for a column is not allowed if the column participates in any type of table constraint (PRIMARY KEY, UNIQUE, REFERENTIAL) or index (INDEX, UNIQUE INDEX).

Change of data type for a column is not allowed if the column is used by a view, procedure, function or trigger.

If a UNIQUE constraint is added to the table, it must be stored in a databank with the TRANSACTION or LOG option.

If a REFERENTIAL constraint is added to the table, both the referencing table and the referenced table must be stored in a databank with the TRANSACTION or LOG option.

If any record exists for a table it is not allowed to add a column with PRIMARY KEY or UNIQUE constraint.

A column of LARGE OBJECT (i.e. BLOB, CLOB, or NCLOB) data type is not allowed in any type of table constraint.

Examples

```
ALTER TABLE staff ADD city VARCHAR(50);

ALTER TABLE staff ALTER COLUMN city SET DATA TYPE NCHAR VARYING(50)
    COLLATE english_1;
```

Notes

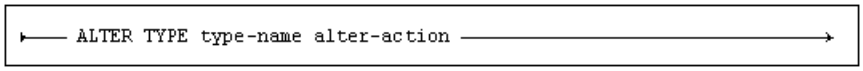
See *Appendix C Limits* for information on the maximum length of a row in a table.

Standard Compliance

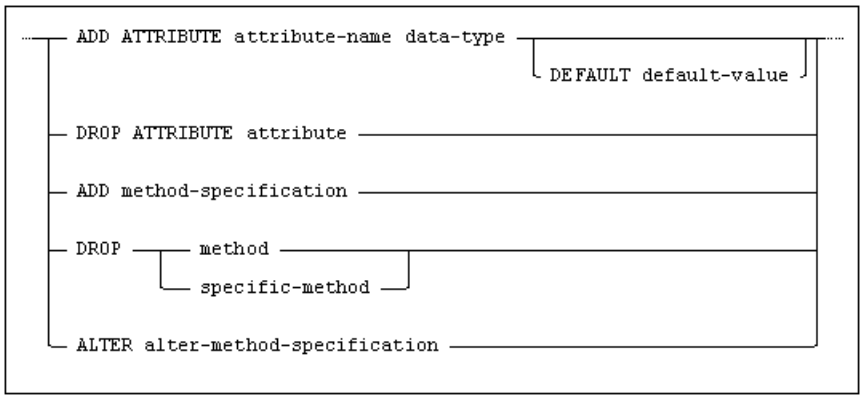
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F033, “ALTER TABLE statement: DROP COLUMN clause” Feature F191, “Referential delete actions”. Feature F251, “Domain support”. Feature F381, “Extended schema manipulation”. Feature F382, “Alter column data type”. Feature F491, “Constraint management”, support for named constraints. Feature T591, “UNIQUE constraints of possibly null columns”. Feature F690, “Collation support”. Feature F701, “Referential update actions”. Feature F721, “Deferrable constraints”, only for referential constraints.
	Mimer SQL extension	The WITH/WITHOUT CHECK clause is a Mimer SQL extension. The SET COMPRESS and SET PAGESIZE clauses are Mimer SQL extensions.

ALTER TYPE

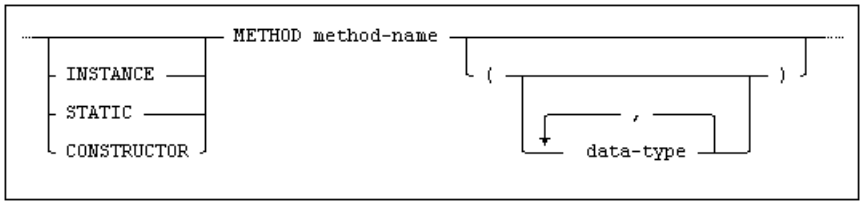
Alter a user-defined type.



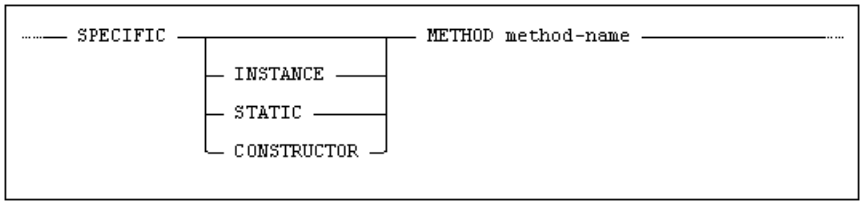
where `alter-action` is:



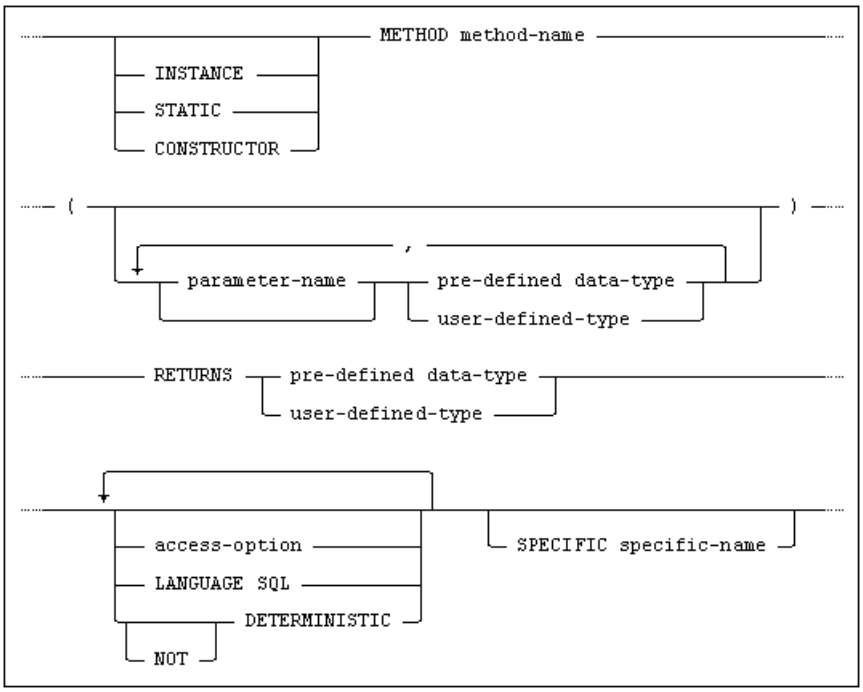
where `method` is:



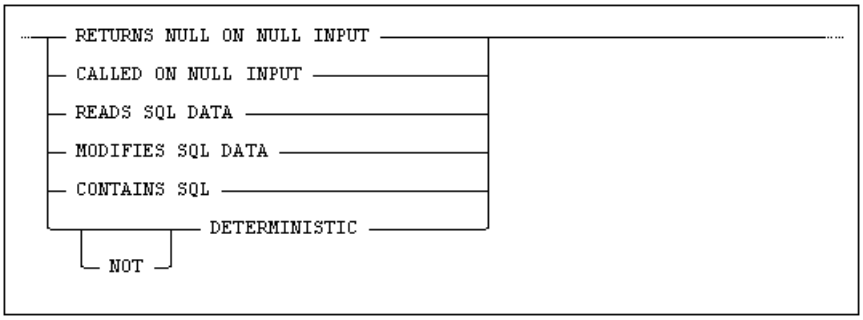
and where `specific-method` is:



and `method-specification` is:



and `routine-attribute` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC

Description

The `ALTER TYPE` statement is used to add method specifications to a type, or to drop method specifications from a type.

It is only the creator of a type that can alter it.

For information on method-specification, please see *CREATE TYPE* on page *298*.

Restrictions

The restrictions for `CREATE TYPE` applies to `ALTER TYPE` also. (See *CREATE TYPE* on page *298*.)

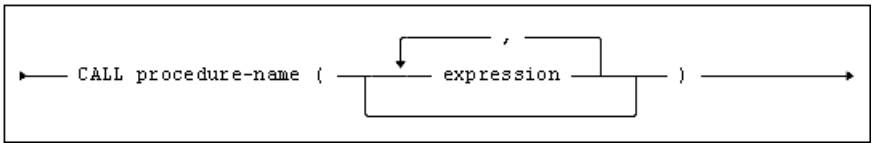
Standard Compliance

Alter a user-defined type.

Standard	Compliance	Comments
SQL-2016	Features outside core	

CALL

Calls a procedure.



Usage

Embedded, Interactive, Module, ODBC, Procedural, JDBC.

Description

The `CALL` statement is used to invoke a procedure. As there can be multiple procedures with the same name, the number of parameters and their type is used to determine which actual procedure should be invoked.

The nature of each `expression` depends on the parameter it applies to. For parameters with mode `OUT` or `INOUT`, `expression` must be a target-variable, see *Target Variables* on page *43*. For parameters with mode `IN`, `expression` may be a value-expression.

The value of `expression` must be assignment-compatible with the data type of the parameter to which it is applied, see *Assignments* on page *77*.

Restrictions

In programming environments a cursor is needed when invoking result set procedures, see *ALLOCATE CURSOR* on page *196* and *DECLARE CURSOR* on page *309* for information about calling result set procedures.

In interactive SQL, the `CALL` statement is used to invoke all types of procedures.

Recursion is permitted, an error will be raised if the internal recursion limit is exceeded.

In a procedural usage context, the called procedure must have an `access-clause` which is lower or equal to that of the calling procedure, see *CREATE PROCEDURE* on page *271* for details about procedure access clause values.

Notes

The `CALL` statement is not used to invoke functions.

Examples

```
CALL PROC1();

CALL PROC2(X,Y);

CALL IDENT1.PROC7(CURRENT_DATE, X+3, Z);
```

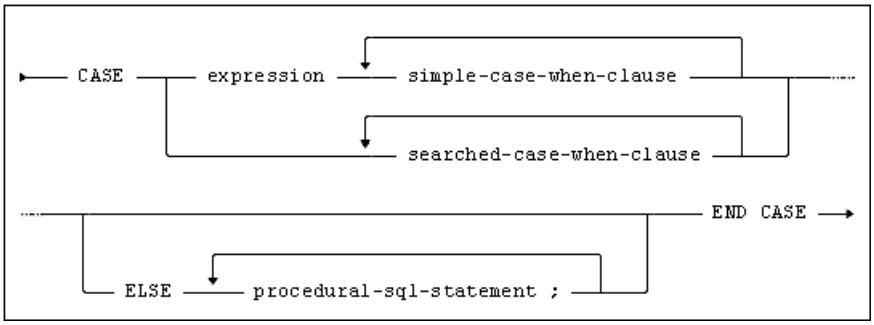
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

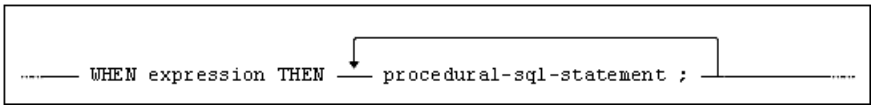
CASE

Allows sequences of SQL statements to be selected for execution based on search or comparison criteria.

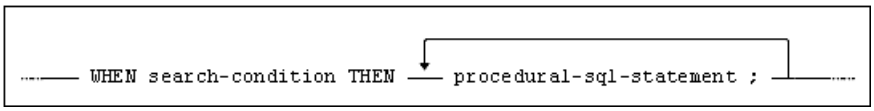
Note: A `CASE` *statement* is not the same as a `CASE` *expression*. A `CASE` statement is for conditional execution of SQL statements, while a `CASE` expression has a return value, and is typically used in SQL queries. See *CASE Expression* on page *145*.



where `simple-case-when-clause` is:



where `searched-case-when-clause` is:



Usage

Procedural.

Description

The `CASE` statement provides a mechanism for conditional execution of SQL statements. It exists in two forms: the simple case and the searched case.

The simple case involves an equality comparison between one expression and a number of alternative expressions, each following a `WHEN` clause.

The searched case involves the evaluation for truth of a number of alternative search conditions, each following a `WHEN` clause.

In each form of the `CASE` it is the first `WHEN` clause to evaluate to true, working from the top down, that determines which sequence of SQL statements will be executed.

There may be one or more SQL statements following the `THEN` clause for each `WHEN`.

If none of the `WHEN` clauses evaluates to true, the SQL statements following the `ELSE` clause are executed. If none of the `WHEN` clauses evaluates to true and there is no `ELSE` clause, an exception condition is raised to indicate that a case was not found.

Providing an ELSE clause supporting an empty compound statement will avoid an exception condition being raised, in cases where no 'else' action is required, when none of the WHEN alternatives evaluates to true.

case-expression

For information on the case-expression, which provides a mechanism for conditionally selecting values, see *CASE Expression* on page *145*.

procedural-sql-statements

For a list of procedural-sql-statements, see *Procedural SQL Statements* on page *193*.

Notes

Flow of control leaves the CASE statement as soon as the SQL statements following the selected THEN, or the ELSE, have been executed (i.e. there is no fall-through as is found in a case statement in, for example, the C programming language).

Examples

Simple CASE statement:

```
DECLARE Y INTEGER;

CASE Y
  WHEN 1 THEN ...
  WHEN 2 THEN ...
  WHEN 3 THEN ...
  ELSE  ...
END CASE;
```

Searched CASE statement:

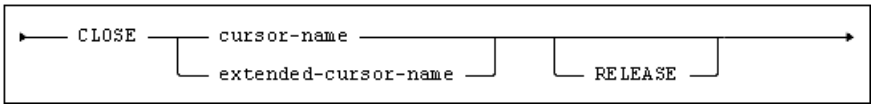
```
CASE
  WHEN EXISTS (SELECT * FROM BILL) THEN ...
  WHEN X > 0 OR Y = 1 THEN ...
ELSE ...
END CASE;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".

CLOSE

Closes a cursor.



Usage

Embedded, Module, Procedural.

Description

The current state of the named cursor is closed.

If any states of the cursor have been saved on a stack by successive `OPEN` statements, see *OPEN* on page *378*, the most recently saved cursor state is restored. Information about whether there are cursor states remaining on the stack is returned as diagnostic information. Otherwise the cursor is closed (deactivated), and may not be used until it has been re-opened with a new `OPEN` statement.

If the optional keyword `RELEASE` is used, all resources allocated to the cursor including any stacked references are destroyed. The cursor must be re-prepared (in dynamic SQL) and reopened before it can be used again.

See *ALLOCATE CURSOR* on page *196* for a description of `extended-cursor-name`.

Restrictions

In a procedural usage context, a cursor cannot be specified by `extended-cursor-name`.

Notes

For the statement to be valid, the cursor in question must be open.

Example

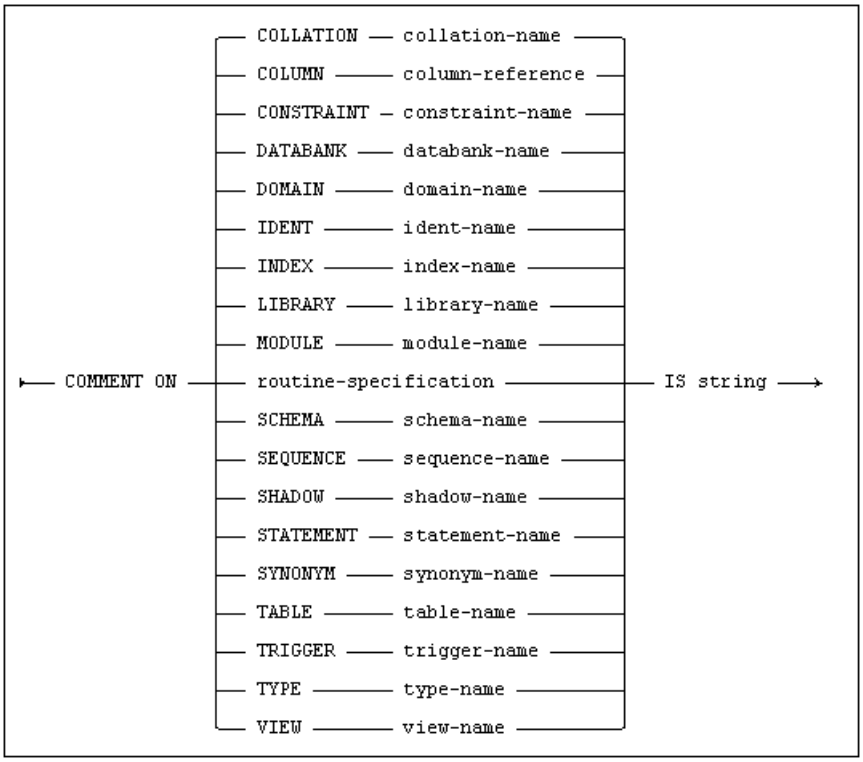
```
DECLARE c_1 CURSOR FOR SELECT product, producer, format,
                                    …
…
…
OPEN c_1;
…
…
CLOSE c_1;
```

Standard Compliance

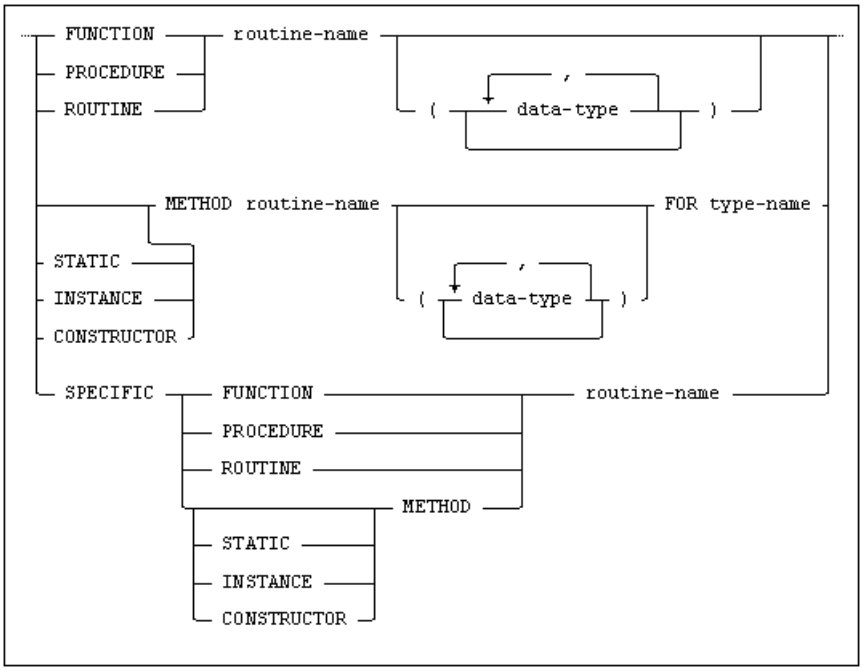
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
	Mimer SQL extension	The keyword RELEASE is a Mimer SQL extension.

COMMENT

Inserts or replaces a comment string on a database object.



where `routine-specification` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The comment `string` for the specified object is stored in the data dictionary. Any previously defined comment for the object is replaced by the new.

Restrictions

A comment can only be stored by the creator of the object.

Notes

A comment string may have a maximum length of 254 characters, and must be enclosed in string delimiters.

Comments may not be altered or dropped directly. However, since the `COMMENT` statement replaces any existing comment with the new text, a comment may be altered simply by issuing a new `COMMENT` statement. A comment may be effectively dropped by issuing a `COMMENT` statement with an empty string.

When a comment is written for a column, the column name must be qualified by a table-name or a view-name in the form `table-name.column-name` or `view-name.column-name`.

Example

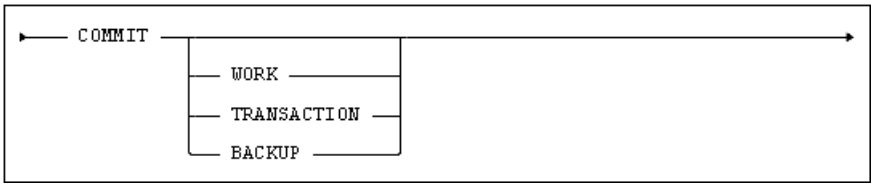
```
COMMENT ON TABLE countries IS 'All countries we ship to';
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	Support for the COMMENT statement is a Mimer SQL extension.

COMMIT

Commits the current transaction.



Usage

Embedded, Interactive, Module, Procedural.

Description

The current transaction is terminated. Database alterations requested in the transaction build-up are executed against the database, provided that no transaction conflict is detected and that no deferred constraints are unsatisfied.

If the commit statement fails, no changes are made in the database, and an error code is set. A transaction conflict causes the SQLSTATE 40000 being raised while an unsatisfied deferred constraint causes the SQLSTATE 40002 being raised.

All cursors opened by the current connection are closed, except cursors that are defined `WITH HOLD`.

Cursors that are defined `WITH HOLD` remain open, but the cursor is no longer positioned on a row. A `FETCH` statement is required to position the cursor on a row before another `DELETE CURRENT` or `UPDATE CURRENT` statement can be executed.

If there is no currently active transaction, any cursors opened by the current ident are closed (except `WITH HOLD` cursors), but the `COMMIT` statement is otherwise ignored. No error code is returned in this case.

Committing a `BACKUP` transaction performs online backup for all databanks for which a `CREATE BACKUP` command has been performed since `START BACKUP`. Please note that this command may be lengthy if backups for large databank files are made.

Restrictions

The `COMMIT` statement cannot be used in a result set procedure because this would close the cursor which is calling it.

The `COMMIT` statement cannot be used within an atomic compound SQL statement, see *COMPOUND STATEMENT* on page *243*.

The `COMMIT BACKUP` statement can only be used when a corresponding `START BACKUP` command has been given. The `COMMIT BACKUP` statement is not supported in procedural mode.

Notes

See the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security,* for a detailed discussion of transaction control.

Example

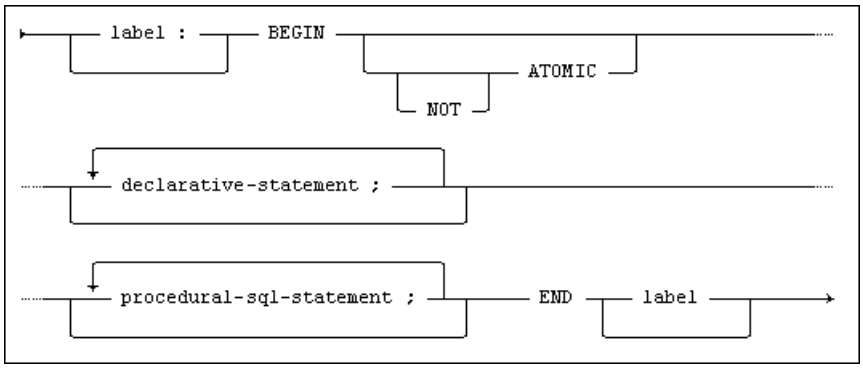
```
EXEC SQL SET TRANSACTION START EXPLICIT

LOOP
   EXEC SQL FETCH C1 INTO :var1,:var2,...,:varn;
   DISPLAY var1,var2,...,varn;
   PROMPT "Update row?";
   EXIT when answer = "yes";
END LOOP

EXEC SQL START;
EXEC SQL UPDATE table SET ...
         WHERE col1 = :var1,
               col2 = :var2, ...
EXEC SQL COMMIT;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
	Mimer SQL extension	Support for the BACKUP and TRANSACTION keywords is a Mimer SQL extension.

COMPOUND STATEMENT

The compound statement (BEGIN/END) is used either in a routine or trigger, or as a separate statement, to create an environment within which variables, cursors, exception condition names and exception handlers can be declared.



Usage

Embedded, Interactive, Module, Procedural.

Description

The procedural SQL statements in a compound statement are executed in sequence whenever the compound statement is executed.

The compound statement may be used wherever a single procedural SQL statement is permitted. Thus, it provides a mechanism for executing a sequence of statements in places where the syntax rules permit only a single statement to be specified.

Compound statements can be nested and the optional label value can be used to qualify the names of objects declared within the compound statement. The label value can also be used in conjunction with the LEAVE statement to control the execution flow by exiting from the compound statement.

A compound statement can be defined as atomic by specifying ATOMIC next to the BEGIN keyword.

When a compound statement is defined as atomic, an 'atomic execution context' becomes active while it, or any invoked routine, is executing. While an atomic execution context is active it is not possible to explicitly terminate a transaction, i.e. the statements START, COMMIT or ROLLBACK are not allowed. Within an atomic compound statement it is possible to declare an undo handler for exception handling. If an undo handler is activated due to an exception, the handler will undo any insert, delete or update operations done within the atomic execution context. If there is no appropriate undo handler found the exception handling will be the same as in a non-atomic context, only the operations performed by the statement causing the exception will be undone.

Restrictions

If ATOMIC is specified, the ROLLBACK and COMMIT statements must not be used in the compound statement.

A compound statement which contains a declaration of an UNDO exception handler must be ATOMIC.

Notes

A compound statement without an ATOMIC or NOT ATOMIC specification is assumed to be NOT ATOMIC.

The value of label must be the same at both ends of the compound statement.

If label is specified at the end of the compound statement it must also be specified at the beginning.

If the LEAVE statement is to be used to exit the compound statement, the label at the beginning must be specified.

Example

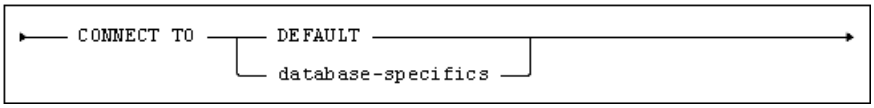
```
CREATE PROCEDURE exproc(IN P_SIRE VARCHAR(30)) MODIFIES SQL DATA
S0: BEGIN
    ...
    S1: BEGIN
        DECLARE EOF BOOLEAN DEFAULT FALSE;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET EOF = TRUE;
        DECLARE HORSES CURSOR FOR
            SELECT *
            FROM HORSES
            WHERE SIRE = P_SIRE;
        DECLARE HORSE AS (HORSES);
        L1: LOOP
            FETCH FROM HORSES INTO HORSE;
            IF  EOF THEN
                LEAVLE L1;
            END IF;
--
-- atomic compound to ensure that both or none of the DML operations are done
--
            BEGIN ATOMIC
                DECLARE UNDO HANDLER FOR SQLEXCEPTION BEGIN END;
                UPDATE HORSE_PEOPLE ...;
                UPDATE HORSE_EVENTS ...;
            END;
        END LOOP;
    END S1;
END S0;
```

Standard Compliance

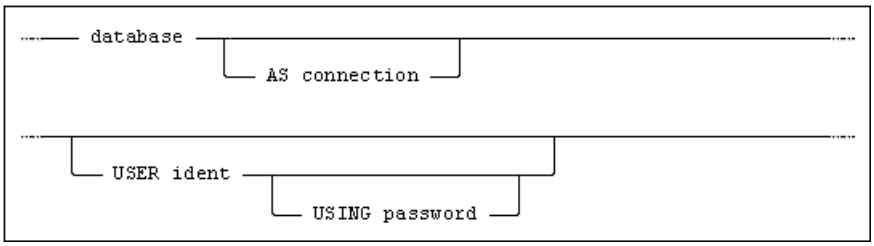
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, “Computational completeness”.

CONNECT

Connects a user ident to a database.



where `database-specifics` is:



Usage

Embedded, Interactive, Module.

Description

The ident is logged into the specified database. The database can exist on the local machine, a local database, or on another machine in a network configuration, a remote database.

The `database`, `connection`, `ident` and the `password` can be supplied either using a host variable or as a literal value.

If an empty string is specified for `database`, a connection is established to the `DEFAULT` database, see the *Mimer SQL System Management Handbook, Chapter 3, The Default Database*, for details on how the `DEFAULT` database is defined in the Mimer SQL system.

If ident is not specified (or ident is specified as blank), the name of the current operating system user is assumed. If the (implicitly or explicitly) specified ident has an `OS_USER` login that matches the current system username, the connection is established without checking the password. See *ALTER IDENT* on page *212* for more details on how to add an `OS_USER` login for an ident.

Note: It is only possible to establish a connection to a remote database without specifying `ident` (or specifying a blank `ident`) if both the node on which the database resides and the node from which the connection is attempted are running the Windows operating system, and the `NamedPipes` protocol is used for the network communication.
It is not possible for the database server node to determine the name of the operating system user currently using the remote machine in other network configurations.

When connected, the ident is able to access the database and becomes the current ident (i.e. the name the returned by `SESSION_USER`).

If `connection` is specified, the name must be a valid identifier or an empty string.

Note: Connection names must be unique. If an empty string is specified, or if no connection name is given, the value of `database` will be used as the connection name.

Password and connection are case sensitive in the CONNECT statement.

Ident and database are not case sensitive in the CONNECT statement.

See *SQL Identifiers* on page *38* for more information.

Restrictions

Only idents of type USER can connect to a database using the CONNECT statement.

Notes

If it is desired that a CONNECT TO DEFAULT be effectively performed, but with the possibility of specifying one or more of connection, ident or password, then specify database-specifics but supply an empty string for database.

The maximum length of database, connection, ident and password is 128 characters.

If an SQL statement is executed in an application without first executing a CONNECT statement, an implicit CONNECT TO DEFAULT is performed.

This requires that a USER ident with an OS_USER login with the same name exist in the default database with the same name as the operating system user, and that the default database either be a local database or a remote database residing on a node which allows the name of the current operating system user to be determined – see the related note in the Description section above for details.

Such an implicit default connection will only be established if the CONNECT statement has not been previously executed in the application. This means that if an explicit connection has been previously established and then disconnected, any subsequent attempt to execute an SQL statement without a current connection will result in either a Connection does not exist error or a transaction rollback depending on the context of the SQL statement.

If only the implicit default connection has been previously established and then disconnected, any subsequent attempt to execute an SQL statement without a current connection will result in that connection being re-established.

Observe that it is possible for the implicit default connection to exist but not be currently active (this will be the case if a connection has been subsequently established and then disconnected).

We recommend that Mimer SQL applications always establish explicit connections and reliance on the implicit default connection is discouraged.

Earlier versions of Mimer SQL used a different syntax for the CONNECT statement, see *Appendix D Deprecated Features*. This syntax is still supported for backward compatibility, but its use is not recommended in new applications.

Example

The following example connects the user JOE to proddatabase using the password hopPsan7:

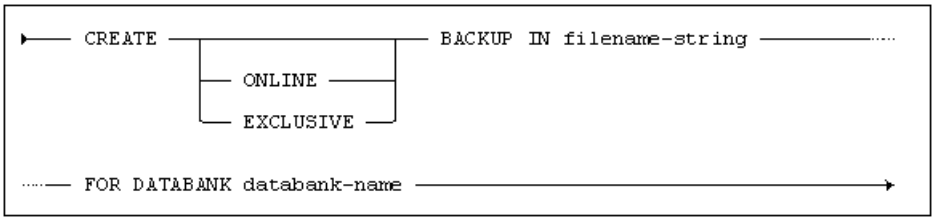
```
CONNECT TO 'proddatabase' USER 'JOE' USING 'hopPsan7';
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F771, “Connection management”.
	Mimer SQL extension	The USING password clause is a Mimer SQL extension.

CREATE BACKUP

Takes a backup copy of a databank file.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

This SQL statement is used to take a backup of a databank.

A backup is a copy of the current databank file and may be used as the basis for a databank recovery operation, see *ALTER DATABANK RESTORE* on page *205*.

The backup will be recorded in a file on disk, the name of the file is specified in the `CREATE BACKUP` statement.

In order to preserve the consistency of the backup between related databanks, a backup of each of the databanks must be taken at exactly the same point in time, from the point of view of transactions updating the databanks. This is done by starting a transaction for the online backup operations using the `START BACKUP` statement (see *START* on page *424*), then executing a `CREATE BACKUP` statement for each databank to be backed up. Finally conclude the transaction by executing the `COMMIT BACKUP` statement (see *COMMIT* on page *241*), or `ROLLBACK BACKUP` statement (see *ROLLBACK* on page *396*.)

It is recommended that all databanks (including system databanks) in a database are backed up together in this way.

The `CREATE BACKUP` command creates the backup file. The actual copying of data from the databank to the backup file is not done until a `COMMIT BACKUP` is executed.

When the keyword `EXCLUSIVE` is used, the backup of the databank will be taken without allowing any concurrent operations. Otherwise, the backup will be taken online, i.e. other operations can be executed concurrently.

When a backup of `LOGDB` is taken, changes made on all databanks are copied to the backup. I.e. this corresponds to taking an incremental backup of all databanks. The entire log is dropped when the backup transaction is committed.

When `LOGDB` is not included in the backup, only the information that applies to the backed up databanks is dropped from the database log. Note that, in this case, it will not be possible to restore the databanks from a previous backup, as the log records are not saved. Therefore, it is highly recommended to always include `LOGDB` whenever any databank is backed up.

Restrictions

CREATE BACKUP requires that the current ident be the creator of the databank or have BACKUP privilege.

The CREATE BACKUP statement cannot be executed unless a transaction, that was started by executing a START BACKUP statement, is currently active.

A backup requires read access to all tables in the databank. It is therefore not possible to take a backup when commands, such as ALTER TABLE and CREATE INDEX, are executing. When a backup has been initiated, commands that require exclusive access will get an error indicating the table is in use by another user.

Notes

The value of filename-string must always be enclosed in string delimiters.

The maximum length of filename-string is 256 characters.

Refer to *Specifying the Location of User Databanks* on page *13* for details concerning specification of the path name components in filename-string.

The CREATE BACKUP command can be used with all databanks in a database including SYSDB, TRANSDB, LOGDB and SQLDB.

The databank option will affect the backup copy:

- **LOG**

  A consistent backup is made of the databank. Transaction logging is used and it will be possible to redo operations made after the backup.

- **TRANSACTION**

  A consistent backup is made of the databank. But as transaction logging is not used, it will not be possible to redo operations made after the backup. I.e. if a disk is corrupted, it is only possible to revert to the state of the latest backup.

- **WORK**

  An online backup of the databank will give a backup which is not completely consistent as the system uses the transaction system to make backups. For a completely consistent backup to be made, the keyword EXCLUSIVE must be used in the CREATE BACKUP command.

- **READ ONLY**

  For read only databanks the backup is always consistent.

The removal of records from the database log to maintain consistency with the backups is handled automatically by these statements, i.e. no additional commands are needed.

Example

The following example starts a backup transaction, creates backup files for the specified databank files, commits the backup and exits:

```
START BACKUP;
   CREATE BACKUP IN 'user_databank' FOR DATABANK user_databank;
   CREATE BACKUP IN 'logdb_backup' FOR DATABANK logdb;
   CREATE BACKUP IN 'sysdb_backup' FOR DATABANK sysdb;
   CREATE BACKUP IN 'transdb_backup' FOR DATABANK transdb;
COMMIT BACKUP;
EXIT;
```

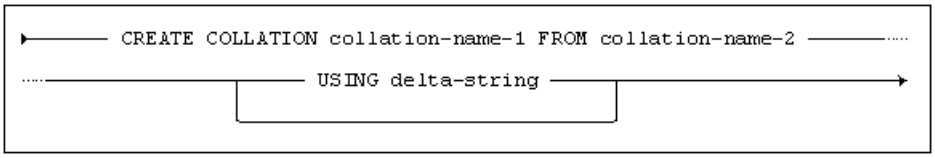
For more information, see the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The CREATE BACKUP statement is a Mimer SQL extension.

CREATE COLLATION

Creates a new collation, based on an existing collation.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A new collation is created. A collation is always based on an existing collation, i.e. the new collation `collation-name-1` is based on the already existing collation `collation-name-2`. If the `CREATE COLLATION` statement has a `USING` clause, the `delta-string` is appended to the definition of the collation specified in the `FROM` clause.

By omitting the `USING` clause, the `CREATE COLLATION` statement can be used to create copies of already existing collations.

See *Tailorings* on page *27* for information about the `delta-string`.

Restrictions

Any ident, who owns a schema, is authorized to create collations.

Notes

Usage privilege on the collation is granted to `PUBLIC`.

Examples

This example will create a Basque collation based on `EOR_1`, where `Ñ` is treated as a separate letter, sorted directly after `N`:

```
CREATE COLLATION basque FROM eor_1 USING '& N < ñ <<< Ñ'
```

The following example will create an English collation based on `ENGLISH_1`, which also sorts numerical data:

```
CREATE COLLATION english_numeric_1 FROM english_1 USING '[NUMERIC ON]'
```

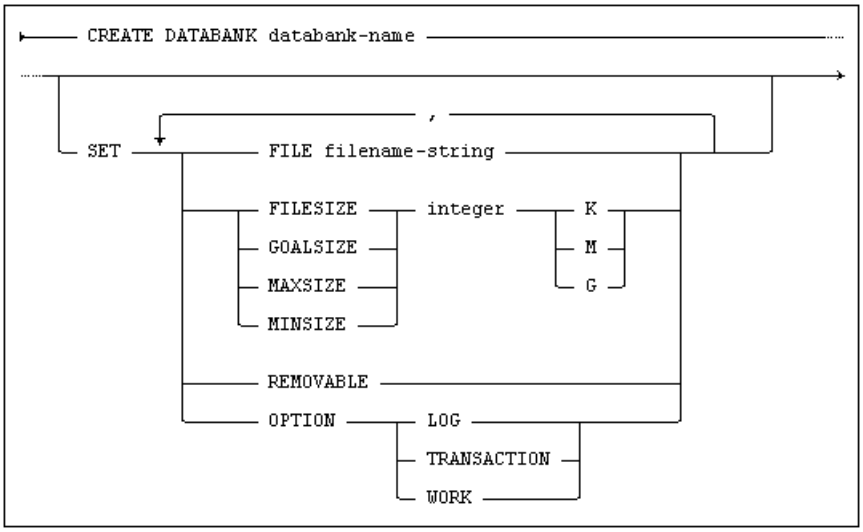
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F690, “Collation support” support for CREATE COLLATION statement.

Standard	Compliance	Comments
	Mimer SQL extension	The syntax with USING delta-string is a Mimer SQL extension. The SQL-2016 syntax contains a FOR <character set> clause, which is not supported in Mimer SQL.

CREATE DATABANK

Creates a new databank.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A new databank is created, i.e. a physical file is created in the host file system and formatted for use as a Mimer SQL databank. Databank attributes that can be set are described below.

FILESIZE

The initial file size can be specified by using the `FILESIZE` option. A value of 2 000 kB is assumed if an initial file size is not specified.

When specifying sizes, `K` (kilo) means that the size (in bytes) is multiplied by 1 024, `M` (mega) means the size is multiplied by 1 048 576, and `G` (giga) means that the size is multiplied by 1 073 741 824.

GOALSIZE

By specifying a `GOALSIZE` value, the system will always try to keep the databank size limited to the value specified.

MAXSIZE

It is possible to specify the maximum file size by using the `MAXSIZE` option.

MINSIZE

It is possible to specify the minimum file size by using the `MINSIZE` option.

This attribute may be useful when executing `ALTER DATABANK DROP FILESIZE`, to assure that the databank file is not shrunk too much. See *ALTER DATABANK* on page *200*.

FILE

The `filename-string` specifies the name of the new databank file in the host file system and this is stored in the data dictionary as the location of the databank file. If a filename is not specified, it will be the same as `databank-name` (and the databank will be created in the database home directory.) The `filename-string` may be represented as character literal, national character literal, or unicode character literal.

OPTION

The databank is created with the transaction and logging options as specified:

LOG

All operations on the databank are performed under transaction control. All transactions are logged, i.e. it will be possible to restore the databank from a backup.

TRANSACTION

All operations on the databank are performed under transaction control. No transactions are logged. (This databank option is assumed if one is not explicitly specified.)

WORK

All operations on the databank are performed without transaction control (even if they are requested within a transaction) and they are not logged. Set operations (`DELETE`, `UPDATE` and `INSERT` on several rows) which are interrupted, will not be rolled back. All secondary indexes created in the databank are flagged as `not consistent`. A secondary index that is flagged as `not consistent` will not offer optimal performance when used in a query, see *UPDATE STATISTICS* on page *432* for information on how to ensure that secondary indexes are consistent.

REMOVABLE

When a databank is set to the `REMOVABLE` attribute, the database system does not signal an error when a `SELECT`, `UPDATE`, or `DELETE` operation is performed on a table in the databank. Instead, the system behaves as if the table is empty and signals an end-of-table condition. If the databank does not have the `REMOVABLE` attribute, an open file error is returned whenever it is accessed and the file cannot be accessed. `INSERT` operations will always signal an error if the databank is inaccessible.

This functionality is useful, for example, if the databank is located on a flash memory card.

Note: A database can be set in `AUTOUPGRADE` mode, which has precedence for `REMOVABLE`, meaning that for a databank having both `AUTOUPGRADE` and `REMOVABLE` enabled a missing databank and/or table will be created. I.e. the file is created whenever it is accessed. If the create fails, the `REMOVABLE` attribute is used.

See *ALTER DATABASE* on page *207* for more information about `AUTOUPGRADE`.

Restrictions

`CREATE DATABANK` requires that the current ident has `DATABANK` privilege.

The databank name must not be the same as that of an existing databank or shadow.

The databank must be created with either the `TRANSACTION` or `LOG` option if any of the following are true:

- the databank is to be shadowed
- the databank will be used to store tables defined with foreign or unique keys
- the databank will be used to store tables that are referenced in a foreign key context
- the databank will be used to store tables holding `UNIQUE` indexes
- the databank contains tables that will accept updates in their primary key column(s)

Notes

For databanks with option `TRANSACTION` and `LOG` the system treats the maximum size as an advisory limit. This limit may be temporarily exceeded. The reason for this is that the actual updating of the databank files are performed in the background while the detection of the maximum size is performed when the applications perform insert operations during transaction buildup. In addition, when several concurrent users are inserting data the actual space is not reserved until the background updates are made.

The creator of the databank is granted `TABLE` privilege on the new databank, with the `WITH GRANT OPTION`.

The value of `filename-string` must always be enclosed in string delimiters. The maximum length of the filename string is 256 characters.

Refer to *Specifying the Location of User Databanks* on page *13* for details concerning the specification of path name components in `filename-string`.

Example

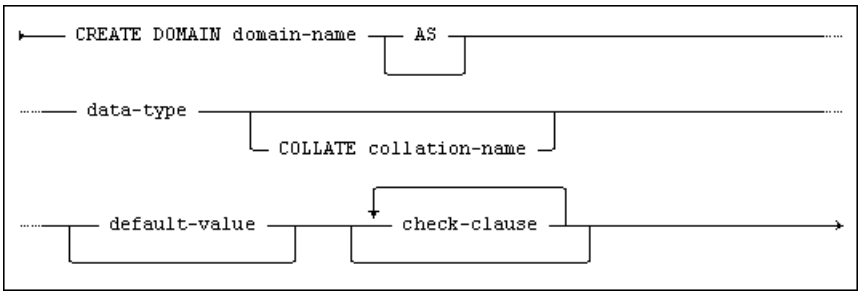
```
CREATE DATABANK mimer_store SET FILE 'mimer_store.dbf',
    FILESIZE 10M, GOALSIZE 10M, MAXSIZE 100M,
    OPTION LOG;
```

Standard Compliance

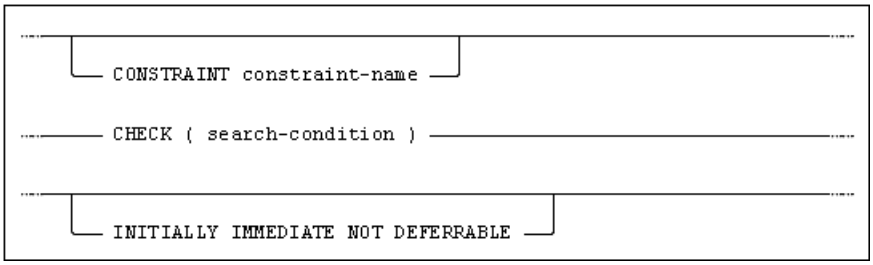
Standard	Compliance	Comments
	Mimer SQL extension	The CREATE DATABANK statement is a Mimer SQL extension.

CREATE DOMAIN

Creates a domain.



where `check-clause` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A domain is created with the properties specified in the statement. Domains may be used instead of explicit data type specifications to define column formats in the `CREATE` and `ALTER TABLE` statements.

If `domain-name` is specified in its unqualified form, the domain will be created in the schema which has the same name as the current ident.

If `domain-name` is specified in its fully qualified form (i.e. `schema-name.domain-name`) the domain will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

Refer to *Data Types in SQL Statements* on page *44* for a description of how the various data types are specified for the domain.

If `default-value` is specified, this value will be assigned to a column defined using the domain whenever a new table row is created or an existing table row is updated without an explicit value being specified for that column.

The COLLATE Clause

If the `COLLATE` clause is specified, the data controlled by the domain will be ordered and compared according to the collation specified.

For more information, see the *Mimer SQL User's Manual, Chapter 4, Collations*.

The CHECK Clause

Specification of a CHECK clause means that only values for which the search condition does not evaluate to false may be assigned to a column defined using the domain.

The search condition, see *Search Conditions* on page *165*, in the CHECK clause may only reference the domain (by using the keyword VALUE), literals, user-defined function invocations or the keyword NULL. The CHECK clause must not contain any non-deterministic expressions, e.g. CURRENT_DATE.

References to columns, subqueries, set functions or host variables are not allowed.

Specifying INITIALLY IMMEDIATE NOT DEFERRABLE explicitly states that the check constraint will be, by default, verified at the time the relevant data manipulation operation is performed rather than when the transaction is committed and that the verification may never be explicitly deferred until the time the transaction is committed. This is also the default behavior. (This is to allow for future extensions to the Mimer SQL syntax.)

Language Elements

default-value, see *Default Values* on page *76*.

Restrictions

An ident must have USAGE privilege on the domain in order to use it.

Notes

The domain name may not be the same as the name of any other domain or used defined type belonging to the same schema.

The CREATE DOMAIN statement does not verify that any specified default value conforms to the restrictions of any specified CHECK clause. It is, therefore, possible to create a domain definition where attempts to store the default value in a column defined using the domain will fail.

Examples

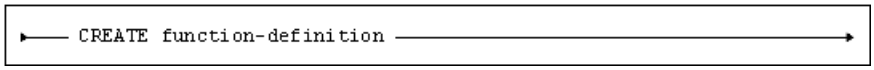
```
CREATE DOMAIN domi AS INTEGER
  CHECK (VALUE IN (-1,0,3) OR VALUE BETWEEN 75 AND 99)

CREATE DOMAIN name AS NCHAR VARYING(48) COLLATE english_1
  CHECK (CHARACTER_LENGTH(VALUE) > 0)
```

Standard Compliance

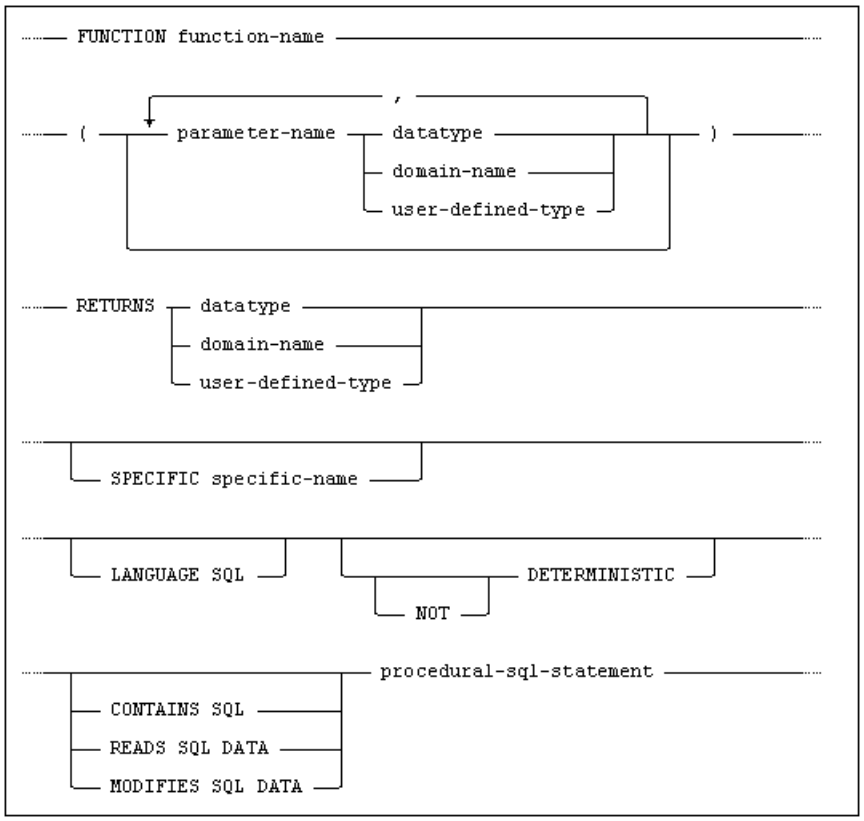
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F251, “Domain support”.

CREATE FUNCTION

Creates a new stored user-defined function.



where `function-definition` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The `function-name` should follow the normal rules for naming database objects, see *Naming Objects* on page *39*.

If `function-name` is specified in its unqualified form, the function will be created in the schema which has the same name as the current ident.

If `function-name` is specified in its fully qualified form (i.e. `schema-name.function-name`) the function will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

The fully qualified function name must be used by all idents except the ident that has the same name as the schema to which the function belongs.

It is possible to create multiple functions with the same name if they differ with regard to either the number of parameters or the data type for the parameter. It is not possible to have multiple functions that only differ with regard to the return data type. See *Mimer SQL Programmer's Manual, Chapter 11, Parameter Overloading* for more information. Type precedence lists are found in *Appendix H Type Precedence Lists*.

Each function can be given a specific name, which must be unique within a schema. If no specific name is given, the system will generate a unique name. The specific name for a function can be retrieved by using the `INFORMATION_SCHEMA` views.

A specific name can be used in `DROP`, `GRANT` and `REVOKE` statements. It is particularly useful when dealing with function with parameter overloading. Instead of having to specify a list of data types, in order to distinguish the function, the specific name can be used.

The `parameter-name` should follow the normal rules for naming SQL identifiers, see *SQL Identifiers* on page *38*.

The permitted data types are pre-defined data types (described in *Data Types in SQL Statements* on page *44*).

If neither `DETERMINISTIC` nor `NOT DETERMINISTIC` is specified, then `NOT DETERMINISTIC` is implicit.

If `DETERMINISTIC` is specified, then the function is guaranteed to produce the same result every time it is invoked with the same set of input values and repeated invocations of it can, therefore, be optimized.

The following access options may be specified:

- **CONTAINS SQL**

  The function may not contain any data-manipulation-statements. All other `procedural-sql-statements` are permitted. The function may only invoke `CONTAINS SQL` functions and procedures. This option effectively prevents a routine from performing read or write operations on data in the database.

- **READS SQL DATA**

  All `procedural-sql-statements` are permitted except those performing updates (i.e. `DELETE`, `INSERT` and `UPDATE`). The function may only invoke `CONTAINS SQL` or `READ SQL DATA` functions and procedures.

  This option effectively prevents a routine from performing write operations on data in the database.

- **MODIFIES SQL DATA**

  All `procedural-sql-statements` are permitted and any function or procedure may be invoked from this type of function.

  This option allows a routine to perform read and write operations on data in the database.

If neither `CONTAINS SQL`, `READS SQL DATA` nor `MODIFIES SQL DATA` is specified, then `CONTAINS SQL` is implicit.

Restrictions

A function created this way cannot be added to a module.

It is possible to create multiple functions with the same name in a schema if the functions have a different number of parameters or parameters with different data types. It is not possible to have multiple functions that only differs with respect to the return data type.

It is not possible to create a synonym for a function name.

A parameter name must be unique within the function.

The parameter mode cannot be specified for a function parameter (as it is for a procedure parameter).

The ROW data type cannot be specified in `data-type`.

If `DETERMINISTIC` is specified, the procedural SQL statement of the function may not contain, or be, a reference to: `SESSION_USER`, `CURRENT_PROGRAM`, `CURRENT_DATE`, `LOCALTIME`, `LOCALTIMESTAMP` or `BUILTIN.UTC_TIMESTAMP` and the function may not invoke functions or procedures that are not deterministic.

If an invoked function attempts to execute a `COMMIT` or `ROLLBACK` statement in a context where this is not permitted, (i.e. after being invoked from within a result set procedure, from within an atomic compound statement or from a data manipulation statement in one of these contexts) an exception will be raised.

An ident must have `EXECUTE` privilege on the function in order to invoke it.

Notes

A function is invoked by specifying its name and parameter list where a value-expression would be used.

All function parameters have the default mode (which is IN). See *CREATE PROCEDURE* on page *271* for details on the parameter modes.

A `parameter-name` can be the same as the name of the function.

If a parameter is defined as using a domain, any input value for this parameter will be verified to ensue that any check constraint is not violated.

Refer to the *Mimer SQL User's Manual, Chapter 7, Creating Functions, Procedures, Triggers and Modules* for details on using the `CREATE FUNCTION` statement in BSQL, where the `@` delimiter is required.

Examples

```
CREATE FUNCTION mimer_store_book.authors_name(p_name VARCHAR(48))
   RETURNS VARCHAR(48)
-- Formats a name into <surname>[,<initial>]
DETERMINISTIC
BEGIN
   DECLARE v_length, v_offset INTEGER;
   DECLARE v_fnm, v_name VARCHAR(48);

   SET v_length = POSITION(',' IN p_name);
   IF v_length = 0 THEN
      SET v_name = UPPER(TRIM(SUBSTRING(p_name FROM 1)));
   ELSE
      -- Append first initial to surname
      SET v_name = UPPER(TRIM(SUBSTRING(p_name FROM 1 FOR v_length)));
      SET v_fnm = UPPER(TRIM(SUBSTRING(p_name FROM v_length+1)));
      SET v_name = v_name || SUBSTRING(v_fnm FROM 1 FOR 1);
   END IF;

   RETURN v_name;
END  -- of routine mimer_store_book.authors_name
```

Example on how to create and use a simple function converting to Celsius degrees from Fahrenheit degrees:

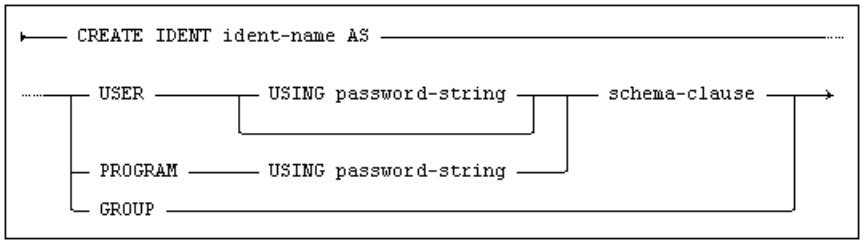
```
CREATE FUNCTION C_from_F (Fdegrees integer) RETURNS integer
    RETURN CAST((Fdegrees - 32) * 5.0 / 9 + 0.5 AS integer);

SELECT C_from_F(temperature) AS Celsius_degrees
FROM US_Weather;

SET ? = C_from_F(451);
```

Standard Compliance

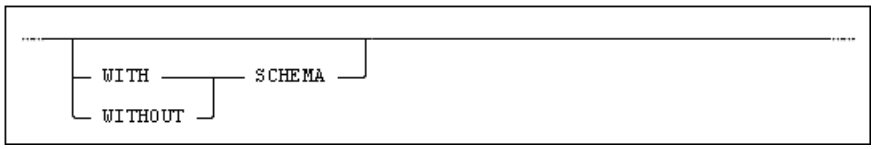
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
	Mimer SQL extension	The possibility to use domains in PSM is a Mimer SQL extension.

CREATE IDENT

Creates a GROUP, PROGRAM or USER (authorization-identity) ident.



where schema-clause is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A new ident is created. If the ident is a USER or PROGRAM ident, a schema with the same name as the ident can also be created. A schema is created by default and when WITH SCHEMA is explicitly specified. For idents who are not supposed to create database objects, it's good practice to specify WITHOUT SCHEMA. (If such an ident later needs a schema, just grant schema to that ident.)

If the ident is a USER, a password can be optionally specified.

If the ident is a PROGRAM ident, a password must be specified.

USER idents are authorized to access a Mimer SQL database by using the CONNECT statement. In interactive contexts, e.g. when Mimer BSQL is started, a USER ident is used to log in.

A USER may connect either by specifying a password or using an OS_USER login. An OS_USER login is added to a USER by using the ALTER IDENT statement. There may be multiple OS_USER logins defined for a USER ident. When a connect statement is executed, the Mimer SQL server will pick up the current system username from the operating system. If there is an OS_USER login for the ident name used in the connect statement that matches the system username there is no need to specify a password in the connect statement. If the system username is the same as the ident name in the Mimer SQL server there is no need to give a ident name when doing a connect statement.

If the connect is done with a tool such as BSQL, this is achieved by entering <return> when prompted for username or password.

PROGRAM idents cannot be used to connect to a database. After a connection has been established (by using a USER or OS_USER ident), the ENTER statement can used to make a PROGRAM ident the current ident. The access rights to the database defined for the PROGRAM ident will thus come into effect.

The ident executing the ENTER statement must have EXECUTE privilege on the PROGRAM ident (the ENTER statement can be executed by a PROGRAM ident).

The ident that executed the ENTER statement will become the current ident again after the LEAVE statement has been executed.

GROUP idents cannot be used to connect to a database. They are used to implement collective authorization of access rights to the database. Other idents become members of a GROUP ident when MEMBER privilege on the GROUP ident is granted to them.

While an ident is a member of a GROUP ident, that ident is effectively granted the privileges held by the GROUP ident.

For a more detailed description of idents, see the *Mimer SQL Programmer's Manual, Chapter 8, Idents and Privileges*.

Restrictions

CREATE IDENT requires that the current ident have IDENT privilege.

The ident must not have the same name as an ident or schema that already exists in the database.

Notes

The creator of a GROUP ident is automatically granted MEMBER privilege on it, with the WITH GRANT OPTION.

The creator of a PROGRAM ident is automatically granted EXECUTE privilege on it, with the WITH GRANT OPTION.

A USER ident password must be at least 1 and at most 128 characters long. A PROGRAM ident password must be at least 1 and at most 18 characters long. A password may contain any characters except space. The case of alphabetic characters is significant. The password string must be enclosed in string delimiters, which are not stored as part of the password.

An ident who is authorized to created new idents (by having IDENT privilege) can also create new schemas.

Example

```
CREATE IDENT mimer_adm AS USER USING 'admin';
```

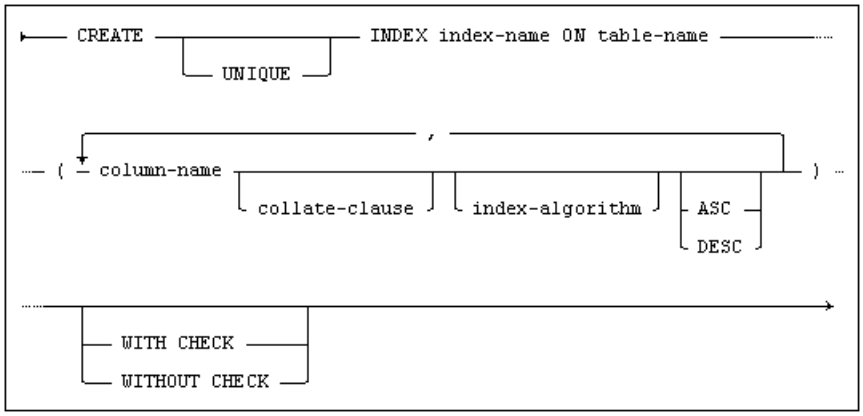
For more information, see the *Mimer SQL User's Manual, Chapter 7, Creating Idents and Schemas*.

Standard Compliance

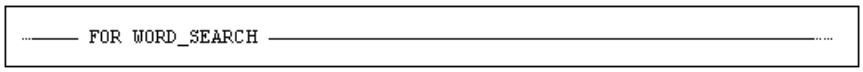
Standard	Compliance	Comments
	Mimer SQL extension	The CREATE IDENT statement is a Mimer SQL extension.

CREATE INDEX

Creates a secondary index on one or more columns of a table.



where `index-algorithm` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A secondary index is created on the column(s) in the table as specified. The index is stored in the data dictionary under the given name. The secondary index is used internally by the optimizer to improve the efficiency of a search.

The UNIQUE Option

If `UNIQUE` is specified each index value (i.e. the value of all index columns together) is only allowed once. In this context two null values are considered equal.

Index-name

If `index-name` is specified in its unqualified form, the index will be created in the schema which has the same name as the current ident.

If `index-name` is specified in its fully qualified form (i.e. `schema-name.index-name`) the index will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

The COLLATE Clause

If the `collate-clause` is specified, the index will be ordered according to the collation specified.

Otherwise, the collation is inherited from the column-definition.

For more information, see the *Mimer SQL User's Manual, Chapter 4, Collations*.

Index-algorithm

If the WORD_SEARCH index algorithm is specified, the index will be optimized for "begins word" searches and "match word" searches. (See *BUILTIN.BEGINS_WORD* on page *93* and *BUILTIN.MATCH_WORD* on page *94*.)

Ascending/Descending

ASC and DESC indicate the sort order of the column within the index. If neither is specified, then ASC is implicit. This makes an index appropriate for queries with a matching ORDER BY specification.

WITHOUT CHECK

The WITH CHECK and WITHOUT CHECK clauses are used to control whether existing table data should be verified for uniqueness or not when a unique index is created. WITH CHECK is the default behavior.

If WITH CHECK is used but the existing data in the table is not unique, the CREATE INDEX statement will fail.

If WITHOUT CHECK is used and the existing data in the table is not unique, the CREATE INDEX statement will still succeed. (After the index has been created, all new data will be verified for uniqueness.)

Note: For a database with the AUTOUPGRADE attribute enabled, the WITHOUT CHECK option must be used when creating a unique index.

See *ALTER DATABASE* on page *207* for more information about AUTOUPGRADE.

Restrictions

An index can not have the same name as a table, view, synonym, constraint or other index in the same schema.

An index must belong to the same schema as the table on which it is created.

Indexes may only be created on base tables, not on views.

UNIQUE indexes may only be created on tables in databanks defined with the LOG or TRANSACTION transaction option.

The WITH/WITHOUT CHECK clause is only valid for unique indexes.

Large object columns (clob, nclob and blob) are not allowed in indexes.

The WORD_SEARCH index algorithm can only be specified for character and national character columns.

The WORD_SEARCH index algorithm may not be specified for unique indexes.

Notes

Each column name must identify an existing column of the table. The same column may not be identified more than once.

Mimer SQL can make use of an index in both the forward and backward direction. It is therefore immaterial whether ASC or DESC is specified if all the index columns have the same sorting direction.

Secondary indexes are automatically maintained and are invisible to the user. The index is used automatically when it provides better efficiency.

Table columns that are in the primary key, a unique key or used in a foreign key reference are automatically indexed (in the order in which they are defined in the key). Therefore, explicitly creating an index on these columns will not improve performance at all.

Consider a table with columns A, B and C of which A and B form the primary key, in that order. An index is automatically created for the column combination (A, B). Therefore, there is no advantage in explicitly creating an index on column A or on the column combination (A, B). Secondary indexes may, however, be advantageous on column B alone or on combinations such as (B, A) or (A, C).

Also, if there is an index on the columns (C, A) there's no need for an index on C alone.

Examples

```
CREATE INDEX cst_date_of_birth ON customers (date_of_birth);

CREATE INDEX cst_ename_french ON customers (ename COLLATE french_1);

CREATE INDEX tracks_track_ws ON tracks (track for word_search);
```

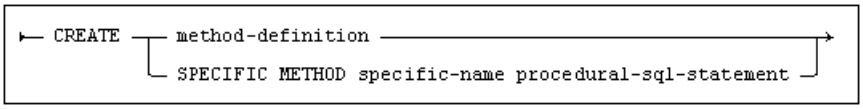
For more information, see *Mimer SQL User's Manual, Chapter 7, Creating Secondary Indexes*.

Standard Compliance

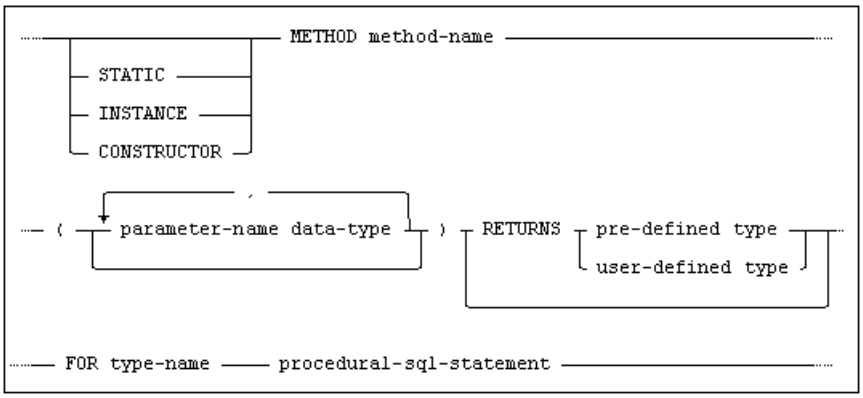
Standard	Compliance	Comments
	Mimer SQL extension	The CREATE INDEX statement is a Mimer SQL extension.

CREATE METHOD

Create a method for a user-defined type.



where `method-definition` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC

Description

Creates a new method. A method returns a single value and can thus be used wherever an expression is allowed. A method is always associated with a user-defined type. Before a method can be created, there must exist a *method specification* for the type with matches the method with regard to parameters and result type. (As it is possible to have method specifications with parameter overloading there must be an exact match.) Method specifications are created using the `CREATE TYPE` statement (see *CREATE TYPE* on page *298*), or using the `ALTER TYPE` statement (see *ALTER TYPE* on page *230*).

If no method type is specified, instance is default. See *Mimer SQL Programmer's Manual, Chapter 13, Invoking Methods*.

The specific name for a method can be retrieved by using the information_schema views.

A specific name can be used in DROP, GRANT and REVOKE statements. It is particularly useful when dealing with routines with parameter overloading. Instead of having to specify a list of data types, in order to distinguish the routine, the specific name can be used.

If no schema name is specified, the method is created in a schema with the same name as the current ident. The ident creating the method must be the owner of the schema. It is only the creator of a user-defined type that can create methods for that type.

The parameter names should follow the normal rules for naming SQL identifiers. All parameters have the parameter mode IN. The data type for a parameter may be a pre-defined type (see *Data Types in SQL Statements* on page *44*) or a user-defined type. The same applies to the result type for the method.

Within the routine body of an instance or constructor method it is possible to use SELF to reference to the actual object that invokes the method.

In a constructor method the attributes have their default values as specified in the `CREATE TYPE` statement.

Restrictions

If the method specification for the method is `DETERMINISTIC` the routine-body may not contain references to `SESSION_USER`, `CURRENT_PROGRAM`, `CURRENT_DATE`, `LOCALTIME`, `LOCALTIMESTAMP` and `BUILTIN.UTC_TIMESTAMP`. It is also not possible to invoke procedures, functions or methods that are deterministic.

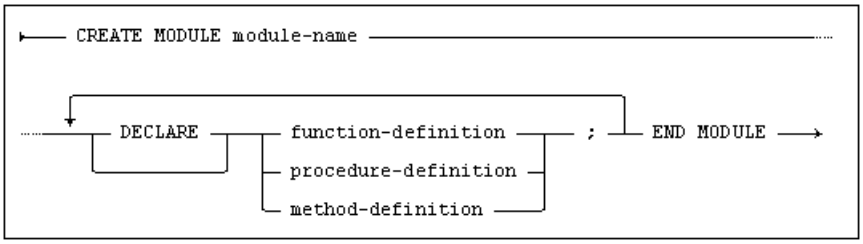
Likewise the access option for the method specification will govern which operations that are allowed.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature S027, “Create method by specific method name”

CREATE MODULE

Creates a new module.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

If `module-name` is specified in its unqualified form, the module will be created in the schema which has the same name as the current ident.

If `module-name` is specified in its fully qualified form (i.e. `schema-name.module-name`) the module will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

A module is simply a convenient enclosure for the collection of one or more routines that are declared as belonging to the module when it is created.

Language Elements

`function-definition`, see *CREATE FUNCTION* on page *258*.

`procedure-definition`, see *CREATE PROCEDURE* on page *271*.

Restrictions

Two modules with the same name cannot belong to the same schema.

All the functions and procedures declared as belonging to the module must be created in the same schema as the module.

Two functions with the same name can only belong to the same schema if they have different numbers of parameters, or the data types for the parameters differ. (See *Mimer SQL Programmer's Manual, Chapter 11, Parameter Overloading* for more information.)

Similarly, two procedures with the same name can only belong to the same schema if they have different numbers of parameters, or the data types for the parameters differ.

It is not possible to create a synonym for a module name.

Notes

The names of the functions and procedures declared as belonging to the module are qualified by using the name of schema to which they belong and not the name of the module.

Example

```
@
CREATE MODULE M1
DECLARE PROCEDURE PROC_1()
READS SQL DATA
BEGIN
...
END;

DECLARE PROCEDURE PROC_2(IN X INTEGER)
MODIFIES SQL DATA
BEGIN
...
END;

DECLARE FUNCTION FUNC_1() RETURNS INTEGER
READS SQL DATA
BEGIN
...
END;


END MODULE
@
```

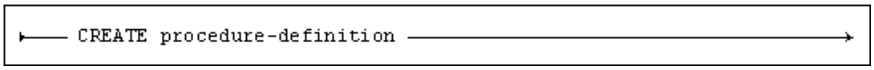
For more information, see the *Mimer SQL User's Manual, Chapter 7, Creating Functions, Procedures, Triggers and Modules*.

Standard Compliance

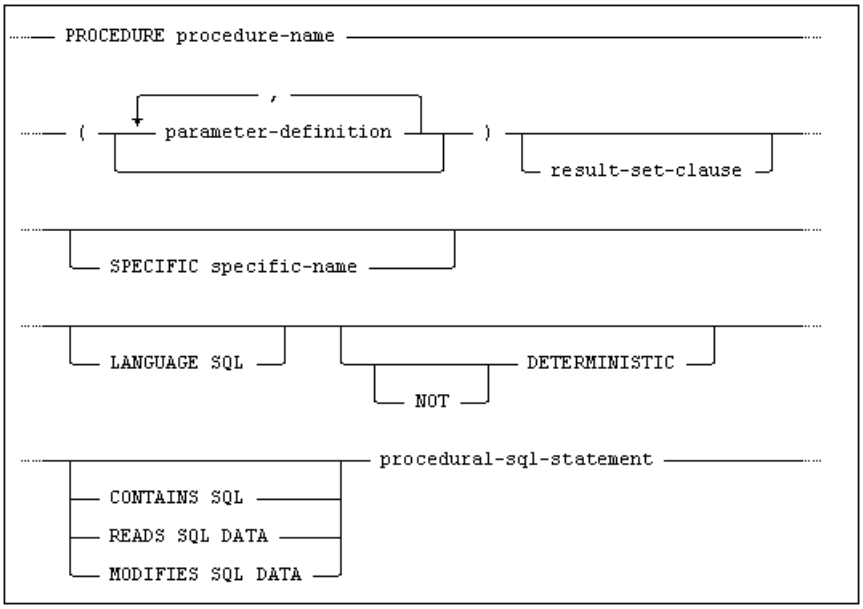
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P001, "Stored modules".

CREATE PROCEDURE

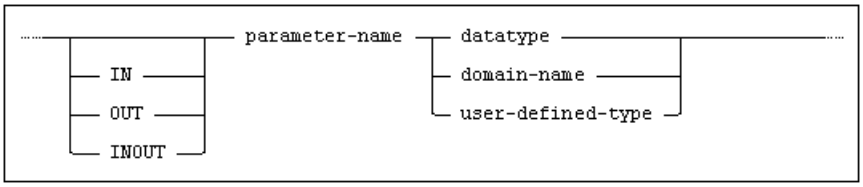
Creates a new stored procedure.



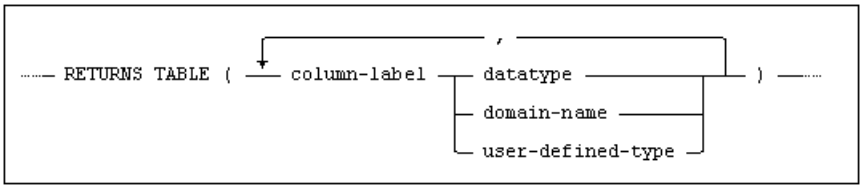
where `procedure-definition` is:



and `parameter-definition` is:



and `result-set-clause` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The `procedure-name` should follow the normal rules for naming database objects, see *Naming Objects* on page *39*.

If `procedure-name` is specified in its unqualified form, the procedure will be created in the schema which has the same name as the current ident.

If procedure-name is specified in its fully qualified form (i.e. schema-name.procedure-name) the procedure will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

The fully qualified procedure name must be used by all idents except the ident that has the same name as the schema to which the procedure belongs.

The parameter-name in the parameter-definition should follow the normal rules for naming SQL identifiers, see *Naming Objects* on page *39*.

It is possible to create multiple procedures with the same name if they differ with regard to either the number of parameters, or the data type for the parameters. See *Mimer SQL Programmer's Manual, Chapter 11, Parameter Overloading* for more information. Type precedence lists are found in *Appendix H Type Precedence Lists*.

Each routine can be given a specific name, which must be unique within a schema. If no specific name is given, the system will generate a unique name. The specific name for a procedure can be retrieved by using the INFORMATION_SCHEMA views.

A specific name can be used in DROP, GRANT and REVOKE statements. It is particularly useful when dealing with procedures with parameter overloading. Instead of having to specify a list of data types, in order to distinguish the procedure, the specific name can be used.

Parameter Definitions

The following mode values may be specified in a parameter-definition:

- IN

  The parameter is effectively read-only, i.e. it cannot be used as the target in an assignment, fetch or select into statement in the procedure

- OUT

  The parameter is effectively write-only, i.e. it can only be used as the target for an assignment and cannot be used in a value expression in the procedure. This type of parameter must be a variable in the procedure CALL statement

- INOUT

  The parameter can be used both as an IN and OUT parameter, this type of parameter must be a variable in the procedure CALL statement.

If neither IN, OUT nor INOUT is specified, then IN is implicit.

The permitted data types, specified in parameter-definition, are pre-defined data types (described in *Data Types in SQL Statements* on page *44*).

Result-Set-Clause

If a result-set-clause is specified, the procedure is created as a result set procedure. A result set procedure is a special type of procedure which returns a result set and is called by being specified in a cursor declaration, see *DECLARE CURSOR* on page *309*, rather than by using the CALL statement.

If neither DETERMINISTIC nor NOT DETERMINISTIC is specified, then NOT DETERMINISTIC is implicit.

If DETERMINISTIC is specified, then the procedure is guaranteed to produce the same result every time it is invoked with the same set of input values and repeated invocations of it can, therefore, be optimized.

Access Options

The following access options may be specified:

- `CONTAINS SQL`

  The procedure may not contain any data-manipulation-statements. All other procedural-sql-statements are permitted. The procedure may only invoke `CONTAINS SQL` functions and procedures.

  This option effectively prevents a routine from performing read or write operations on data in the database.

- `READS SQL DATA`

  All `procedural-sql-statements` are permitted except those performing updates (i.e. `DELETE`, `INSERT` and `UPDATE`). The procedure may only invoke `CONTAINS SQL` or `READ SQL DATA` functions and procedures.

  This option effectively prevents a routine from performing write operations on data in the database.

- `MODIFIES SQL DATA`

  All `procedural-sql-statements` are permitted and any function or procedure may be invoked from this type of procedure.

  This option allows a routine to perform read and write operations on data in the database.

If neither `CONTAINS SQL`, `READS SQL DATA` nor `MODIFIES SQL DATA` is specified, then `CONTAINS SQL` is implicit.

Restrictions

A procedure created this way cannot be added to a module.

It is possible to create multiple procedures with the same name in a schema if the procedures have a different number of parameters or parameters with different data types.

It is not possible to create a synonym for a procedure name.

A parameter name must be unique within the procedure.

The `ROW` data type cannot be specified in `parameter-definition` or in a `result-set-clause`.

A result set procedure may only have parameters with mode `IN`.

A result set procedure or a routine invoked from within a result set procedure, must not execute a `COMMIT` or `ROLLBACK` statement because this would interfere with the cursor used when the result set procedure is called.

If `DETERMINISTIC` is specified, the `procedural-sql-statement` of the procedure may not contain, or be, a reference to: `SESSION_USER`, `CURRENT_DATE`, `CURRENT_PROGRAM`, `LOCALTIME`, `LOCALTIMESTAMP` or `BUILTIN.UTC_TIMESTAMP`.

The option `MODIFIES SQL DATA` cannot be used for a result set procedure.

An ident must have `EXECUTE` privilege on the procedure in order to invoke it.

Notes

Refer to the *Mimer SQL User's Manual, Chapter 7, Creating Functions, Procedures, Triggers and Modules*, for details on using the `CREATE PROCEDURE` statement in Mimer BSQL where the `@` delimiter is required.

If an in parameter is defined as using a domain, any input value for this parameter will be verified to ensue that any check constraint is not violated. If an out parameter is defined as using a domain, the parameter will be initialized with the default value for the domain.

Example

```
CREATE PROCEDURE res_proc (IN A INTEGER, IN B INTEGER)
                 RETURNS TABLE (CLIENT_NAME VARCHAR(32), CLIENT_ID INTEGER)
READS SQL DATA
BEGIN
...
END;
```

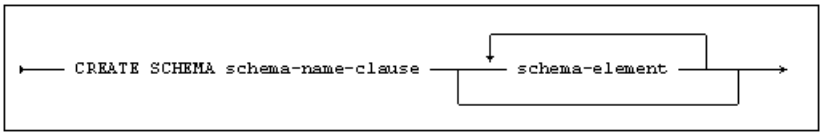
For more information, see the *Mimer SQL User's Manual, Chapter 7, Creating Functions, Procedures, Triggers and Modules*.

Standard Compliance

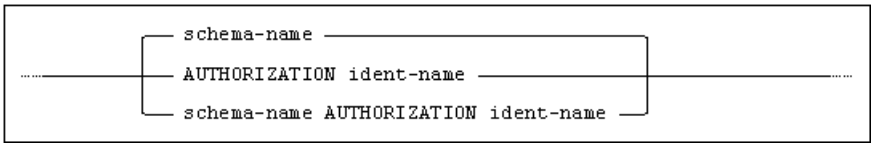
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
	Mimer SQL extension	The result-set-clause is a Mimer SQL extension.
	Mimer SQL extension	The possibility to use domains in PSM is a Mimer SQL extension.

CREATE SCHEMA

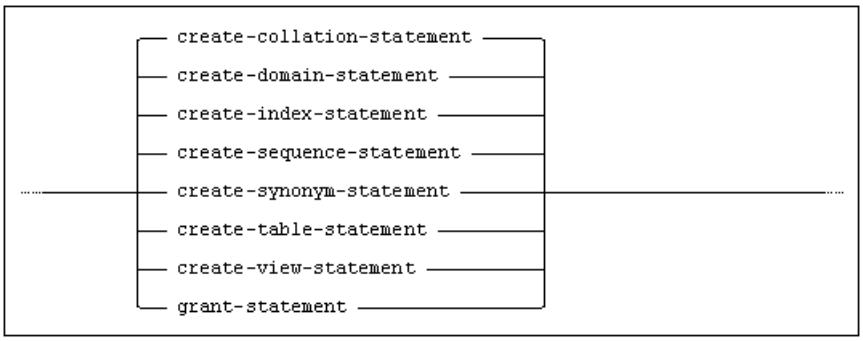
Creates a new schema.



where `schema-name-clause` is:



and `schema-element` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A new schema is created with the name specified in `schema-name-clause`. If `schema-name` is specified, the schema is created with that name, otherwise the name of the schema will be the same as `ident-name`.

If `ident-name` is specified, the schema and all the other objects created by the `CREATE SCHEMA` statement are created with the named ident as the effective current ident.

A `schema-element` is a `CREATE` or `GRANT` statement that is specified using the normal syntax for such a statement and which is executed by the `CREATE SCHEMA` statement in the normal way.

Language Elements

`create-collation-statement`, see *CREATE COLLATION* on page *251*.

`create-domain-statement`, see *CREATE DOMAIN* on page *256*.

`create-index-statement`, see *CREATE INDEX* on page *264*.

`create-sequence-statement`, see *CREATE SEQUENCE* on page *277*.

`create-synonym-statement`, see *CREATE SYNONYM* on page *284*.

`create-table-statement`, see *CREATE TABLE* on page *285*.

`create-view-statement`, see *CREATE VIEW* on page *302*.

`grant-statement`, see *GRANT ACCESS PRIVILEGE* on page *359* or *GRANT OBJECT PRIVILEGE* on page *361*.

Restrictions

The schema name must not be the same as that of a schema which already exists in the database.

`CREATE SCHEMA` requires that the current ident has `SCHEMA` or `IDENT` privilege.

The value for `ident-name` is currently restricted to be the name of the current ident.

If a `schema-element` contains a `CREATE` statement where the name of the object to be created is specified in its fully qualified form (i.e. `schema_name.object_name`), the `schema-name` component must be the same as the name of the schema being created by the `CREATE SCHEMA` statement.

Notes

If a `schema-element` contains a `CREATE` statement where the name of the object to be created is specified in an unqualified form, it will be created in the schema created by the `CREATE SCHEMA` statement (and not the schema with the same name as the current ident as is usual for `CREATE` statements).

It is possible for one `schema-element` to reference the objects created by other `schema-element`'s regardless of the order of creation of the objects. All object references are verified at the conclusion of the `CREATE SCHEMA` statement when all the `schema-element`'s have been executed and all objects have been created.

Example

```
CREATE SCHEMA mimer_store_music;
```

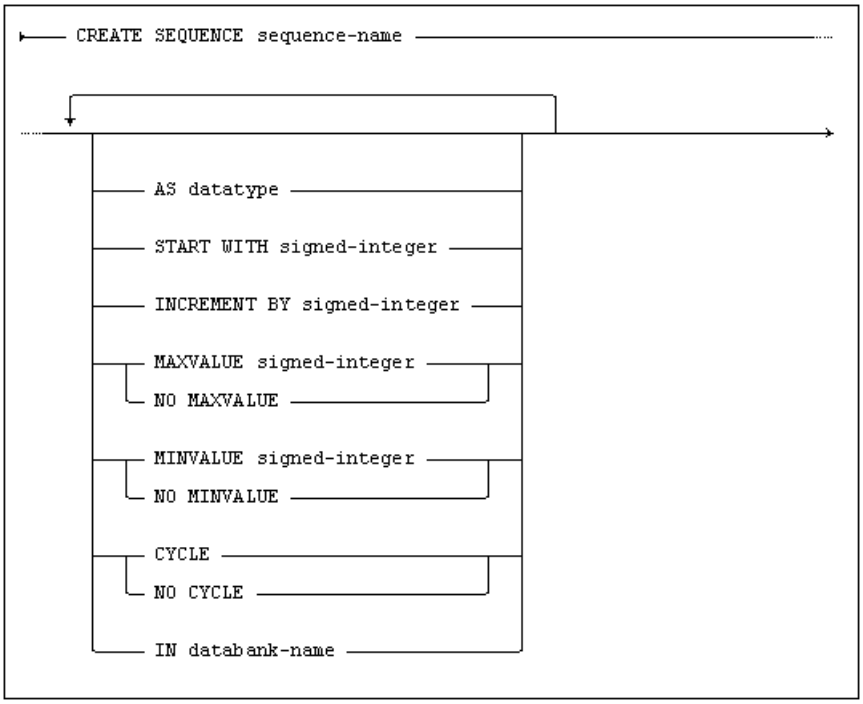
For more information, see the *Mimer SQL User's Manual, Chapter 7, Creating Idents and Schemas*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F171, "Multiple schemas per user" support for schema name-clause. Feature F251, "Domain support" support for CREATE DOMAIN statement in schema definition. Feature F690, "Collation support" support for CREATE COLLATION statement in schema definition.
	Mimer SQL extension	Support for CREATE INDEX and CREATE SYNONYM statements in a schema definition is a Mimer SQL extension.

CREATE SEQUENCE

Creates a new sequence.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A sequence generates a series of exact numeric values by starting at the start value and proceeding in steps as defined by the increment value. The increment can either be positive or negative. If increment is positive the sequence is called an ascending sequence, and if increment is negative it is a descending sequence. The default increment value is 1.

If no start value is specified, the start value for a regular ascending sequence is `MINVALUE`, and for a descending sequence it is `MAXVALUE`.

The default `MINVALUE` is 1. The default `MAXVALUE` is the highest possible value (depends on the data type, see table below).

`MINVALUE`, `MAXVALUE`, start value and increment value must all be between the limits for the data type for the sequence.

Data type	Lowest possible value	Highest possible value
SMALLINT	-32768	32767
INTEGER	-2147483648	2147483647
BIGINT	-9223372036854775808	9223372036854775807

If no data type is specified, INTEGER is default.

Start value must be between MINVALUE and MAXVALUE (if specified).

The set of possible values for a sequence is limited by MINVALUE and MAXVALUE. If CYCLE option is specified for the sequence these values will be generated endlessly, while if NO CYCLE is specified, the sequence will be exhausted once all possible values has been generated. NO CYCLE is the default if cycle option is not specified.

To generate a new value for a sequence the expression

```
next value for sequence-name
```

is used. This can be used in all DML-statements where an expression is allowed. It can also be used in the default clause for a column or for a domain definition. See *NEXT VALUE* on page *111*.

To get the latest generated value within a session the expression

```
current value for sequence-name
```

is used. The generated value is kept for each session. This means that current value is not affected by other users using the same sequence. See *CURRENT VALUE* on page *100*.

The IN databank-name specifies in which databank sequence data should be stored. The user creating the sequence must have SEQUENCE privilege on the databank.

If IN databank-name is not specified, the system will choose a databank on which the user has SEQUENCE privilege. If more than one such databank exists, databanks created by the current ident are chosen in preference to others and the databank with the most secure transaction option is chosen (i.e. a databank with LOG option would be chosen in preference to one with TRANSACTION option).

Restrictions

The sequence-name should follow the normal rules for naming database objects, see Naming Objects.

If sequence-name is specified in its unqualified form, the sequence will be created in the schema which has the same name as the current ident.

If sequence-name is specified in its fully qualified form (i.e. schema-name.sequence-name) the sequence will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

Two sequences with the same name cannot belong to the same schema.

An ident must have USAGE privilege on the sequence in order to use it.

The sequence must be created in a databank on which the current ident has SEQUENCE privilege.

A databank used to store sequences must have TRANSACTION or LOG option.

Notes

The sequence is created with an undefined current value initially. When NEXT VALUE FOR sequence-name is used for the first time after the sequence is created, the initial value for the sequence is returned and established as the current value of the sequence.

If CURRENT VALUE FOR sequence-name is used when the current value of the sequence is undefined, an error will be raised.

Examples

A sequence with default options:

```
create sequence mseq01;
```

When used this sequence will generate values between 1 and 2147483647 in steps of one, starting with the value 1.

A smallint based sequence:

```
create sequence mseq02
      as smallint
      start with 2
      increment by 3
      minvalue 1
      maxvalue 10
      cycle;
```

This sequence will generate the following (repeating) series of values:

2, 5, 8, 1, 4, 7, 10, 1, 4, 7, 10 ...

A bigint based sequence:

```
create sequence mseq03 as bigint
      increment by -1;
```

When used this sequence will generate values between 9223372036854775807 and 1 in descending steps of one.

Use a sequence for column default:

```
create sequence idseq start with 1;
create table orders (orderid integer default next value for idseq,
                     primary key (orderid),
                     purchasedate date,
                     customerid integer references customer);
```

If a new row is inserted into the orders table without specifying a value for the orderid column, the sequence will be used to generate a new unique value for the column.

Note: It is possible that not every value in the series of values defined by the sequence will be generated. In case of a server failure it is possible that some of the values in the series might be skipped.

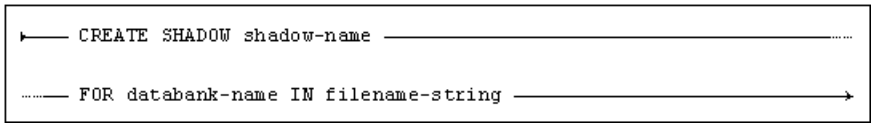
For more information, see the *Mimer SQL User's Manual, Chapter 7, Creating Sequences*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature T176 "Sequence generator support".
	Mimer SQL extension	The IN databank clause is a Mimer SQL extension.

CREATE SHADOW

Creates a new shadow for a databank.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A new shadow for a databank is created, i.e. a new physical file is created in the host file system and the contents of an existing Mimer SQL databank is copied to the file, see the *Mimer SQL System Management Handbook, Chapter 10, Mimer SQL Shadowing,* for more details.

The `filename-string` specifies the name of the new file in the host system and is stored in the data dictionary as the physical location of the shadow.

Restrictions

`CREATE SHADOW` is only for use with the optional Mimer SQL Shadowing module.

The ident executing the `CREATE SHADOW` statement must hold `SHADOW` privilege.

The `CREATE SHADOW` statement cannot be used if the databank to be shadowed is `OFFLINE`.

The databank to be shadowed (specified by `databank-name`) cannot be used by any other user while the shadow is being created.

A databank must have either the `TRANSACTION` or `LOG` option if it is to be shadowed, since the shadowing facility requires transaction handling.

The Mimer SQL shadowing functionality cannot be used together with the automatic upgrade feature. When setting the `AUTOUPGRADE` attribute on a database, following `CREATE SHADOW` statements will fail. And vice versa, when having databank shadows in the system it will not be possible to enable the `AUTOUPGRADE` attribute.

Notes

The `shadow-name` may not be the same as that of any existing databank or shadow.

The value of `filename-string` must always be enclosed in string delimiters. The maximum length of the filename string is 256 characters.

Refer to *Specifying the Location of User Databanks* on page *13* for details concerning the specification of path name components in `filename-string`.

Example

The following example creates the shadow `SYSDB_S` for the Mimer SQL system databank `SYSDB` in the file `sysdb_s`:

```
CREATE SHADOW SYSDB_S FOR SYSDB IN 'sysdb_s'
```

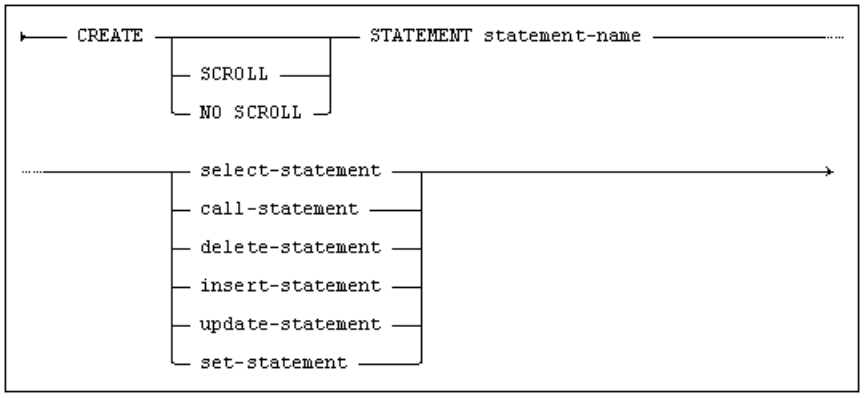
For more information, see the *Mimer SQL System Management Handbook, Chapter 10, Mimer SQL Shadowing*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The CREATE SHADOW statement is a Mimer SQL extension.

CREATE STATEMENT

Stores a precompiled statement in the data dictionary for execution later.



Usage

Embedded, Interactive, Module, ODBC, JDBC

Description

The statement specified, stored with the name `statement-name`, is compiled and optimized and thereafter stored in its compiled form in the data dictionary.

The statement can then be executed through the `EXECUTE STATEMENT` command (see *EXECUTE STATEMENT* on page *335*.) A statement that produces a result set (e.g. `SELECT`), needs a cursor to read it.

You can use the `SCROLL` option to specify that the result set should be read by a scrollable cursor.

If you use the `NO SCROLL` option, the result set will only be accessible for no-scroll cursors.

If you do not specify an option, the result set can be used by both scroll and no-scroll cursors.

Once the precompiled statement has been created, only cursors with the correct scroll mode can be used to read the result set.

Language Elements

`call-statement`, see *CALL* on page *233*.

`delete-statement`, see *DELETE* on page *317*.

`insert-statement`, see *INSERT* on page *368*.

`select-statement`, see *SELECT* on page *398*.

`set-statement`, see *SET* on page *404*.

`update-statement`, see *UPDATE* on page *426*.

Restrictions

The ident executing the `CREATE STATEMENT` command must have adequate access rights to perform the stored command.

Notes

The `statement-name` must not be the same as any other statement name in the schema.

Parameter marker references may be used in the statement to denote values that are supplied when the statement is executed. Parameter markers can either be a `?` or on the form `:variable`.

Examples

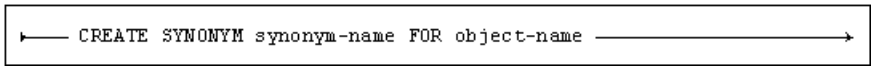
```
CREATE SCROLL STATEMENT seltaba
SELECT col1, col2
FROM taba WHERE col3 < 10

CREATE STATEMENT updtaba
UPDATE taba SET col2 = :inpvar
WHERE col3 < :col3var

CREATE STATEMENT callme
CALL proc1('ABC', ?)
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The CREATE STATEMENT command is a Mimer SQL extension.

CREATE SYNONYM

Creates an alternative name for a table, view or another synonym.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A synonym is created for the table, view or synonym specified in `object-name`.

If `synonym-name` is specified in its unqualified form, the synonym will be created in the schema which has the same name as the current ident.

If `synonym-name` is specified in its fully qualified form (i.e. `schema-name.synonym-name`) the synonym will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

Restrictions

A synonym may only be created if the creator has some access privilege on the object specified in `object-name`.

A synonym can only be created for an existing table, view or synonym.

The synonym name may not be the same as the name of any other table, view, index, constraint or synonym already belonging to the schema in which the synonym is created.

Notes

The synonym is stored in the data dictionary and it may be used to refer to the associated table or view wherever `object-name` would normally be used in the syntax specifications for SQL.

Synonyms are not the same as correlation names. The latter are defined in the `FROM` clause of `select-specifications`, see *Chapter 11, The SELECT Expression*, and apply only within the context of the statement where they are defined.

Example

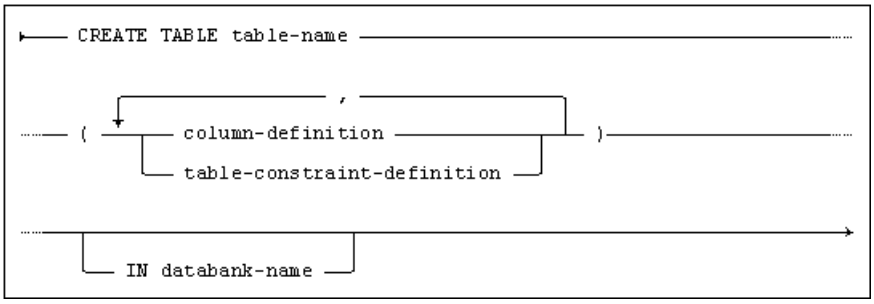
```
CREATE SYNONYM artists FOR mimer_store_music.artists;
```

Standard Compliance

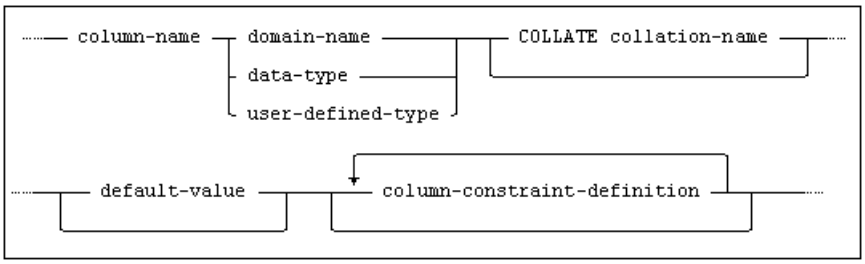
Standard	Compliance	Comments
	Mimer SQL extension	The CREATE SYNONYM statement is a Mimer SQL extension.

CREATE TABLE

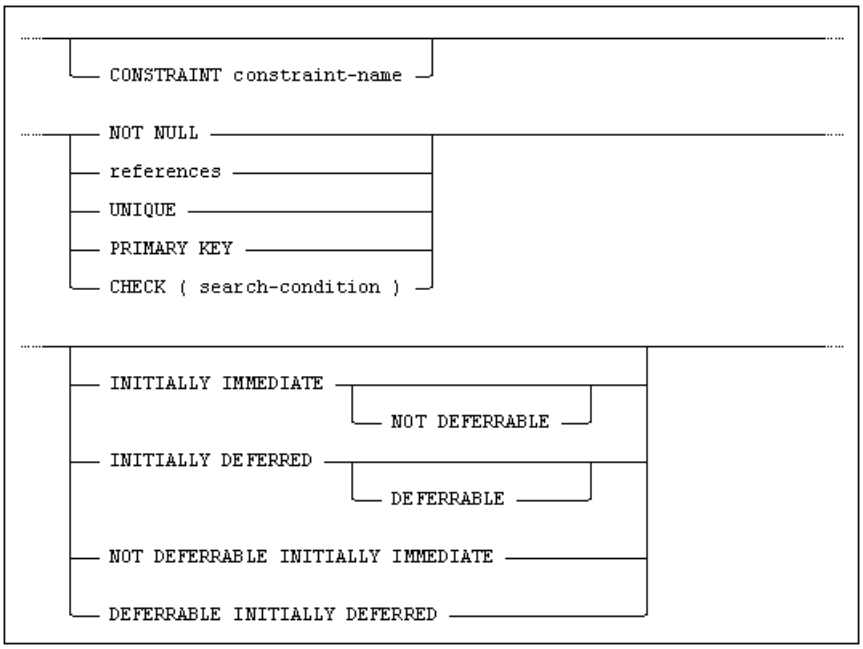
Creates a new table.



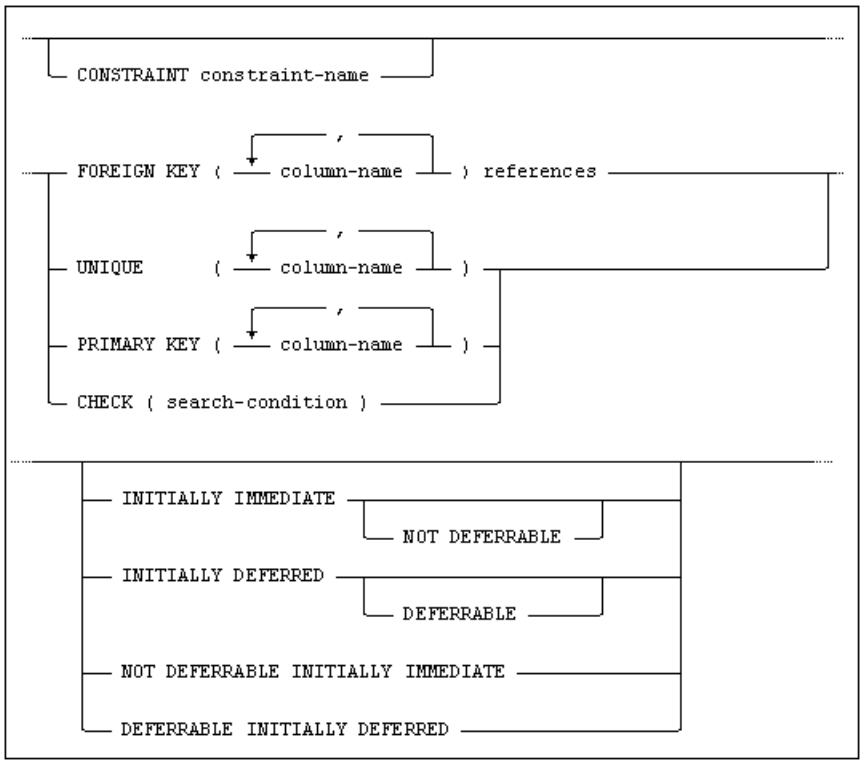
where `column-definition` is:



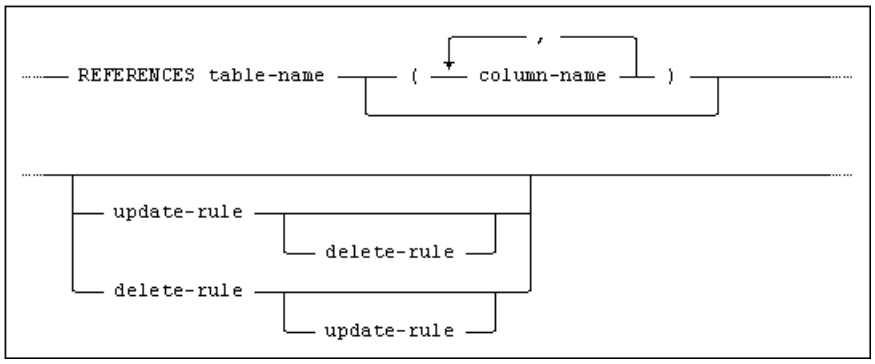
and `column-constraint-definition` is:



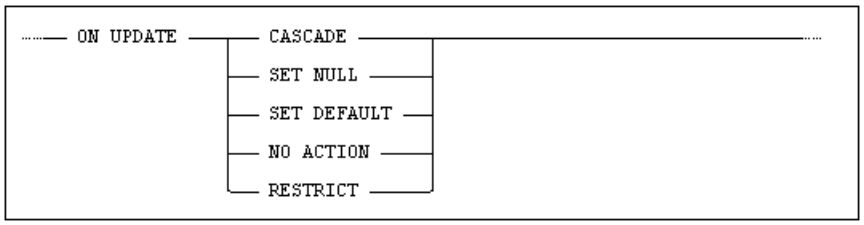
and `table-constraint-definition` is:



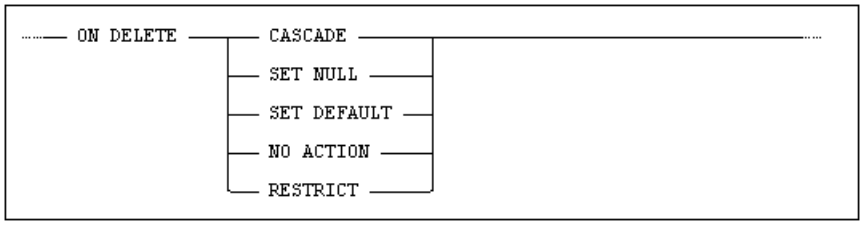
and `references` is:



and `update-rule` is:



and `delete-rule` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A new table is created as specified.

If `table-name` is specified in its unqualified form, the table will be created in the schema which has the same name as the current ident.

If `table-name` is specified in its fully qualified form (i.e. `schema-name.table-name`) the table will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

The table definition includes a list of `column-definition`'s and `table-constraint-definition`'s.

The table must be created in a databank on which the current ident has `TABLE` privilege.

If `IN databank-name` is not specified, the system will choose a databank on which the user has `TABLE` privilege. If more than one such databank exists, databanks created by the current ident are chosen in preference to others and the databank with the most secure transaction option is chosen (i.e. a databank with `LOG` option would be chosen in preference to one with `TRANSACTION` option and one with `TRANSACTION` option in preference to one with `WORK` option).

The new table is empty until data is inserted.

Column Definitions

The columns will appear in the table in the order specified. Each column name must be unique within the table. Column formats may be specified either by explicit data type, see *Data Types in SQL Statements* on page *44*, or by specifying the name of a domain to which the column will belong. In the latter case, all the properties of the domain apply to the column.

A default value can be defined for the column by specifying `default-value` in `column-definition` or by having the column belong to a domain for which a default value is defined. A default value specified in `default-value` will take precedence over a domain default value and the data type of the value specified in `default-value` must conform to the data type of the column.

The default value will be assigned to a column whenever an `INSERT` is performed with no explicit value supplied. If the defined default value does not conform to other constraints, e.g. a `CHECK` constraint, then an `INSERT` must supply a value. The default value will also be assigned by an `UPDATE` statement with `DEFAULT` specified as update value.

The COLLATE Clause

In order to enable string data comparison and ordering, you can specify a `COLLATE` clause for a column.

A collation specified in the column-definition will take precedence over a domain collation.

By doing so, the collation defined will always considered in clauses such as `WHERE`, `ORDER BY` and `GROUP BY`, as well as when using relational and comparison operators. For more information, see the *Mimer SQL User's Manual, Chapter 4, Collations*.

Table Constraints

One or more constraints may be defined on the table, either by specifying a `column-constraint-definition` in a `column-definition` or by the specifying a `table-constraint-definition` in the table element list.

All table constraints may be named by specifying a `constraint-name` in the `column-constraint-definition` or `table-constraint-definition`. If a constraint is defined without specifying an explicit name, an automatically generated name will be assigned to it.

Note: Automatically generated constraint names start with `SQL_`, so it is recommended that this initial character sequence be avoided when explicitly specifying a constraint name.

Constraint names are shown in the appropriate `INFORMATION_SCHEMA` views, see *Chapter 13, INFORMATION_SCHEMA dictionary views*.

The constraint name is used to identify a constraint when it is dropped using the `ALTER TABLE` statement. For more information, see *ALTER TABLE* on page *226*.

NOT NULL Constraints

If this constraint is specified in a `column-constraint-definition` in the `column-definition` for a column, the column will not accept an attempt to insert the null value.

PRIMARY KEY Constraint

One `PRIMARY KEY` can be defined for the table, composed of one or more of the table columns.

The same column must not occur more than once in the primary key.

A column that is a part of the primary key will implicitly be constrained as `NOT NULL`, regardless of any `NOT NULL` constraints explicitly defined on the table. The null value cannot, therefore, occur in a primary key column.

The purpose of a primary key is to define a key value that uniquely identifies each table row, therefore the primary key value for each row in the table must be unique.

The primary key value for a table row is the combined value of the column(s) making up the primary key. The column(s) of the primary key (and their order in the key) can be defined using the `PRIMARY KEY` clause in a `table-constraint-definition`.

If the primary key for the table is to be composed of only a single column, then it can be defined by specifying `PRIMARY KEY` in a `column-constraint-definition` in the `column-definition` for that column.

UNIQUE Constraints

One or more `UNIQUE` constraints can be defined on the table. A `UNIQUE` constraint defines a unique key for the table. A unique key is composed of one or more table columns, just like the primary key. A column must not occur more than once in the same unique key.

A unique key defines a key value that uniquely identifies each row in the table, therefore a table cannot contain two rows which have the same value for a unique key unless one or more of the columns are null.

A unique key must not be composed of the same set of column(s) (occurring in any order) as either the primary key or an existing unique key defined for the table.

A unique key value for a table row is the combined value of the column(s) making up the unique key. The column(s) of the unique key (and their order in the key) can be defined using the `UNIQUE` clause in a `table-constraint-definition`.

If a unique key is to be composed of only a single column, then it can be defined by specifying `UNIQUE` in a `column-constraint-definition` in the `column-definition` for that column.

Note: Multiple occurrences of the null-value do not violate a `UNIQUE` constraint. (However, a UNIQUE INDEX allows just one null-value. See *CREATE INDEX* on page *264*.)

REFERENTIAL Constraints

A referential constraint defines a foreign key relationship between the table being created (the referencing table) and another table in the database (the referenced table).

A foreign key relationship exists between a key (the foreign key) in the referencing table and the primary key or one of the unique keys of the referenced table.

The foreign key in the referencing table is defined by using the `FOREIGN KEY` clause in `table-constraint-definition` and is composed of one or more columns of the referencing table. The same referencing table column cannot occur more than once in the foreign key.

The corresponding key in the referenced table is specified by using the `REFERENCES` clause in `references`. If a list of column names is not specified after the name of the referenced table, then the primary key of the referenced table is assumed.

More than one foreign key can be defined for a table and the same table column can occur in more than one of the foreign keys.

The name of the referenced table must be specified in its fully qualified form if the name of the schema to which it belongs is not the same as the current ident.

The *i-th* column in the referencing table foreign key corresponds to the *i-th* column in the specified key of the referenced table and both keys must be composed of the same number of columns.

The data type and data length of each column in the referencing table foreign key must be identical to the data type and data length of the corresponding column in the specified key of the referenced table.

The effect of a referential constraint is to constrain table data in a way that only allows a row in the referencing table which has a foreign key value that matches the specified key value of a row in the referenced table.

One or more of the columns in a foreign key may permit the null value (this will be the case if there is no `NOT NULL` constraint or equivalent `CHECK` constraint in effect for the column).

A referencing table row which has a foreign key value with the null value in at least one of the columns will always fulfil the referential constraint and therefore be acceptable as a row in the referencing table.

If all of the columns in a foreign key are constrained not to accept the null value, then the only rows that will be accepted in the referencing table are those with a foreign key value that already exists in the corresponding key of the referenced table.

A referential constraint can be defined by specifying a `FOREIGN KEY` clause in `table-constraint-definition`. If a referencing table foreign key is to be composed of only a single column, then the referential constraint can be defined by specifying `references` in a `column-constraint-definition` in the `column-definition` for that column.

Rules can be defined in references that specify an action to be performed on the affected row(s) of the referencing table when a delete or update operation in the referenced table causes a referential constraint to be violated (because rows would consequently exist in the referencing table those foreign key value did not match the corresponding key value of a row in the referenced table).

One of the following actions can be specified for a referential constraint for update operations:

- `ON UPDATE CASCADE` - referencing columns in affected rows in the referencing table will be set to the updated value of the referenced columns in the referenced table.
- `ON UPDATE SET NULL` - referencing columns in affected rows in the referencing table will be set to the null value.
- `ON UPDATE SET DEFAULT` - referencing columns in affected rows in the referencing table will be set to the default value for that column.
- `ON UPDATE RESTRICT` - this implies that the checking of the constraint will be done once for each row affected by the update statement.
- `ON UPDATE NO ACTION` - this implies that the checking of the constraint is done either when the update statement is completed or at commit, depending on the deferrability of the constraint.

If no `update-rule` is specified, `ON UPDATE NO ACTION` is default.

One of the following actions can be specified for a referential constraint for delete operations:

- `ON DELETE CASCADE` - the affected rows in the referencing table are deleted
- `ON DELETE SET NULL` - the relevant foreign key columns of the affected rows in the referencing table will be set to the null value.
- `ON DELETE SET DEFAULT` - the relevant foreign key columns of the affected rows in the referencing table will be set to the default value for that column.
- `ON DELETE RESTRICT` - this implies that the checking of the constraint will be done once for each row affected by the delete statement.
- `ON DELETE NO ACTION` - this implies that the checking of the constraint is done either when the delete statement is completed or at commit, depending on the deferrability of the constraint.

If a `delete-rule` is not specified, then `ON DELETE NO ACTION` is default.

CHECK Constraints

One or more check constraints can be defined on the table, which will determine whether the changes resulting from an INSERT or UPDATE operation will be accepted or rejected.

The search-condition which defines the check constraint must not contain a select-specification, an invocation of a set function, a reference to a host variable, or a non-deterministic expression. (However, as a work-around invoked functions may contain such functionality.)

If the search-condition of a check constraint specified in a table-constraint-definition contains column references, they must be columns in the table being created.

If the search-condition of a check constraint specified in a column-constraint-definition in a column-definition contains a column reference, it must be the column identified by column-name of the column-definition.

The search-condition of a check constraint defined on the table will be evaluated whenever a new row is inserted into the table and whenever an existing row is updated.

The values for any column reference(s) contained in the search-condition will be taken from the row being inserted or updated.

The data change operation will only be accepted if the search condition does not evaluate to false.

Constraint Characteristics

When defining a constraint it is possible to specify that the constraint should either be INITIALLY IMMEDIATE or INITIALLY DEFERRED. This attribute defines when the constraint is checked.

A constraint that is specified as INITIALLY IMMEDIATE is checked when a statement is executed. Constraints are INITIALLY IMMEDIATE by default.

A constraint that is specified as INITIALLY DEFERRED is checked at commit.

Note that for a foreign key constraint, it is only the checking that is deferred to commit time. I.e. referential actions such as CASCADE, SET DEFAULT, SET NULL and RESTRICT are all performed when the statement is executed.

Language Elements

default-value, see *Default Values* on page *76*.

search-condition, see *Search Conditions* on page *165*.

Restrictions

CREATE TABLE requires TABLE privilege on the databank in which the table is to be created.

The table name must not be the same as the name of any other table, view, synonym, index or constraint belonging to the same schema.

If a domain name is specified for column-definition, USAGE privilege must be held on the domain.

Each table-constraint-definition can only be specified once in the CREATE TABLE statement.

If a UNIQUE constraint is defined on the table, it must be stored in a databank with the TRANSACTION or LOG option.

If a REFERENTIAL constraint is defined, both the referencing table and the referenced table must be stored in a databank with the TRANSACTION or LOG option.

A constraint name must not be the same as the name of any other table, view, synonym, index or constraint belonging to the same schema.

The creator of the table must hold REFERENCES privilege on all the columns specified in references.

The name of a view cannot be specified for table-name in references.

When creating a table with a foreign key, you, the creator, must have exclusive access to the referenced table.

A column of LARGE OBJECT data type is not allowed in any type of table constraint but NOT NULL.

The constraint mode INITIALLY DEFERRED can only be specified for referential constraints.

Notes

The creator of the table is granted all access privileges to the table WITH GRANT OPTION.

In a REFERENTIAL constraint, the referenced table can be the same as referencing table. In this situation, the table data is constrained in a way that only allows the foreign key columns to contain key values that are already present in the referenced (primary or unique) key.

If a name is not specified for a table or column constraint, a system generated name is applied to it. System generated names will begin with SQL_ so it is recommended that this starting character sequence be avoided for explicitly specified constraint names.

The primary key and the unique keys for a table are not dissimilar in their function and they constrain data in the same way apart from the fact that primary key columns are always defined as not null, however a unique key should not be used instead of a primary key. One reason for this is that the primary key is handled more efficiently than the unique keys, so there is a performance advantage. See *Relational Databases – Selected Writings* by C. J. Date for a discussion of primary and unique keys.

Example

```
CREATE TABLE eng_table (col_1 INTEGER,
                        col_2 NCHAR(2000) COLLATE english_1,
    PRIMARY KEY (col_1));
```

For many more examples, see the *Mimer SQL User's Manual, Chapter 7, Creating Tables*.

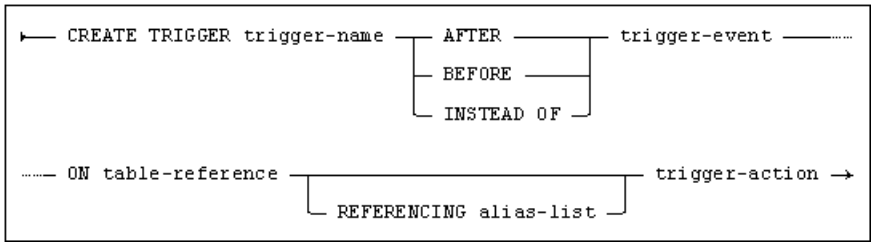
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

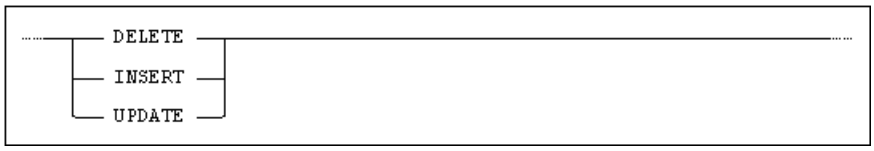
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F191, “Referential delete actions”. Feature F251, “Domain support”. Feature F491, “Constraint management”, support for named constraints. Feature F690, “Collation support”. Feature F701, “Referential update actions”. Feature F721, “Deferrable constraints”, only for referential constraints. Feature T591, “UNIQUE constraints of possibly null columns”.
	Mimer SQL extension	Support for the IN databank-name clause is a Mimer SQL extension.

CREATE TRIGGER

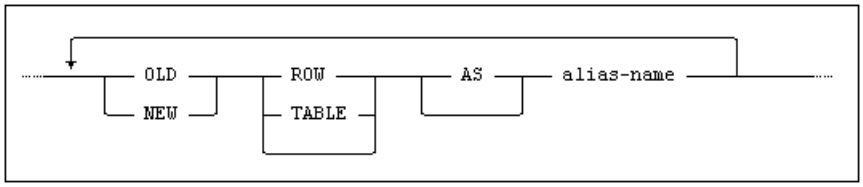
Creates a trigger which is invoked by data changes in a named table or view.



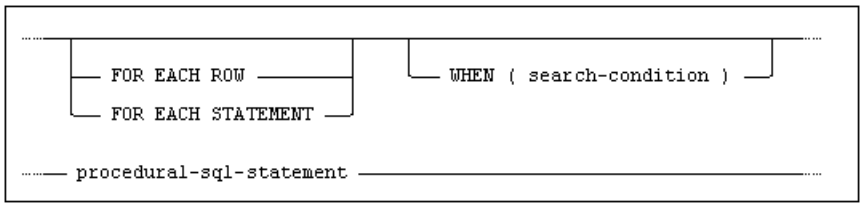
where `trigger-event` is:



and `alias-list` is:



and `trigger-action` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

A trigger is created on a table or view (table reference).

For a complete description of triggers, see the *Mimer SQL Programmer's Manual, Chapter 12, Triggers*.

The `trigger-name` should follow the normal rules for naming database objects, see *Naming Objects* on page *39*.

If `trigger-name` is specified in its unqualified form, the trigger will be created in the schema which has the same name as the current ident.

If `trigger-name` is specified in its fully qualified form (i.e. `schema-name.trigger-name`) the trigger will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

The `trigger-action` will be executed when the data manipulation operation specified by `trigger-event` occurs on `table-reference` and any `search-condition` specified in the `WHEN` clause of the `trigger-action` evaluates to true.

There are two types of triggers, row triggers and statement triggers. A row trigger is executed once for each row that is modified by a data manipulation operation. A statement trigger is invoked once for a data manipulation operation.

A row trigger is defined by specifying for each row in the trigger definition. If for each statement is specified or the for each clause is omitted, the trigger will be a statement trigger. Note that a statement trigger will always be invoked, regardless of if any rows are modified by the data manipulation operation. A row trigger will only be executed if any row is affected by the data manipulation.

The trigger time specifies when a trigger is executed. For a more detailed description of this, see *Mimer SQL Programmer's Manual, Chapter 12, Triggers*.

It is possible to create multiple triggers for the same event and time and if so the triggers will be executed in the order they are created.

In a statement trigger it is possible to refer to temporary tables that contains the data affected by the data manipulation operation. These tables are named in the referencing clause and are commonly referred to as the old and new table. These tables are read only.

The old table shows the data as it were before the data manipulation operation and the new table shows the data after the statement has taken place.

The old table can be used if the trigger event is delete or update. The new table can be used if the trigger event is update or insert. The temporary tables will only be created if there is at least one statement trigger that references the old or new table.

In a row trigger it is possible to refer to the row being affected by the data manipulation operation. The old and new row variables can be seen as implicit parameters for the triggers. The old row variable is read only in all cases but the new row variable can be modified if the trigger time is before. The old and new row are defined as records where each field corresponds to a column in the table reference. To refer to individual fields a dot notation is used. See example below.

If the trigger time is INSTEAD OF the table reference must be a view. This is the only trigger time that can be specified for a trigger defined on a view.

If there is an INSTEAD OF trigger defined for a view this means that the data manipulation operation for a view will not be performed, but the trigger will be executed instead. In the trigger it is possible to do data manipulations on the tables on which the view is defined. Thus it is possible to make any view updatable by creating an instead of trigger. An instead of trigger may also use the old and new tables to access the data affected by the data manipulation operation that caused the trigger to be executed.

Restrictions

The trigger and `table-reference` must belong to the same schema.

Two triggers with the same name cannot belong to the same schema.

If the trigger time is `INSTEAD OF`, then `table-reference` must be the name of a view.

OLD TABLE and NEW TABLE may each be specified only once in the alias-list and the same alias-name must not appear twice in the list.

OLD ROW and NEW ROW may each be specified only once in the alias-list and the same alias-name must not appear twice in the list.

OLD ROW and NEW ROW may only be specified if FOR EACH ROW is specified.

OLD TABLE or OLD ROW may not be specified if the `trigger-event` is INSERT.

NEW TABLE or NEW ROW may not be specified if the `trigger-event` is DELETE.

AFTER and INSTEAD OF are currently not supported for row triggers.

If the trigger time is BEFORE and FOR EACH STATEMENT is specified, the REFERENCING keyword and alias-list must not be specified.

If the procedural-sql-statement of the trigger-action is a COMPOUND STATEMENT, it must be ATOMIC.

The creator of the trigger must hold the appropriate access rights, with grant option, for all operations performed in the trigger action.

The trigger-action must not contain a COMMIT or ROLLBACK statement.

If the trigger time is BEFORE, the following restrictions apply to the trigger-action:

- the trigger-action must not contain any SQL statement that performs data update (i.e. DELETE, INSERT and UPDATE statements are not permitted)
- a routine which possibly MODIFIES SQL DATA may not be invoked from within the trigger-action.

A trigger can be created on tables that have columns defined as LARGE OBJECT data type, with the restrictions that it is not possible to refer to such columns in the new table in an instead of trigger and that it is not possible to modify such fields in the new row variable.

Notes

The `trigger-action` is always executed in the transaction started for the data manipulation operation which caused the trigger to be invoked. Thus, if the data manipulation operation is subject to a rollback, all operations performed in the `trigger-action` will also be undone and an unhandled error occurring in the `trigger-action` will be treated like an error in the triggering data manipulation statement. Situations like this can be handled using condition handlers. (See *DECLARE HANDLER* on page *312*.)

During the execution of the `trigger-action`, the effect of changes made in the transaction are visible.

The scope of the `trigger-action` is the optional `WHEN` clause and the `procedural-sql-statement`.

The tables specified by using `OLD TABLE` and `NEW TABLE` in the `alias-list` are temporary and are local to scope of the `trigger-action`. It is not possible to perform any data change operations on either table and the data contained in each will not otherwise change during the time it exists.

Data manipulation operations performed in the `trigger-action` may cause the trigger to be invoked recursively. Trigger execution in a recursive situation will proceed normally in every respect.

If the body of the trigger contains operations on tables located in a databank with work option, these operations will not be part of the atomic statement that constitute the trigger execution.

If the `procedural-sql-statement` in a trigger contains a compound statement, it is possible to declare condition handlers for handling errors that may occur in the trigger code. See *DECLARE HANDLER* on page *312*.

Examples

```
CREATE TRIGGER mimer_store_book.titles_after_insert
   AFTER INSERT ON mimer_store_book.titles
REFERENCING NEW TABLE AS btl
BEGIN ATOMIC
…
…
END  -- of trigger mimer_store_book.titles_after_insert

CREATE TABLE versions(document_id int, version_date date);

CREATE TRIGGER set_version_date BEFORE UPDATE ON versions
REFERENCING NEW ROW AS new_version OLD ROW AS old_version FOR EACH ROW
    IF  old_version.version_date = new_version.version_date THEN
        SET new_version.version_date = current_date;
    END IF;
```

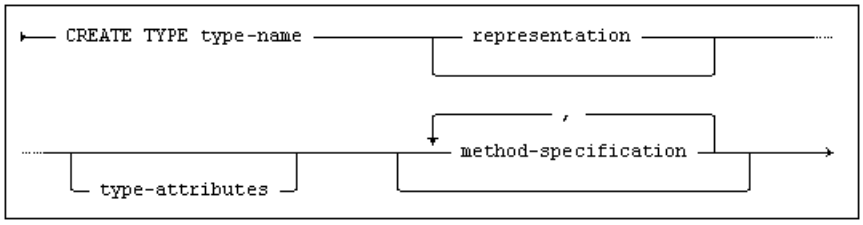
For more information, see the *Mimer SQL Programmer's Manual, Chapter 12, Triggers*.

Standard Compliance

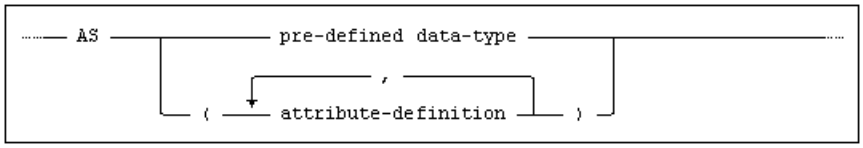
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F571, "Truth value tests". Feature T211, "Basic trigger capability". Feature T212, "Enhanced trigger capability". Feature T213, "INSTEAD OF triggers".

CREATE TYPE

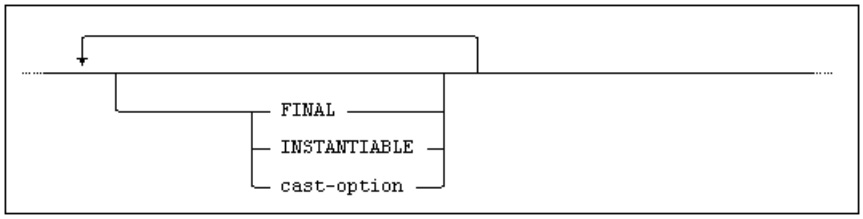
Create user-defined type.



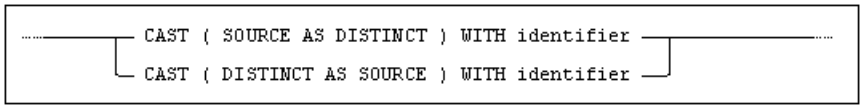
where `representation` is:



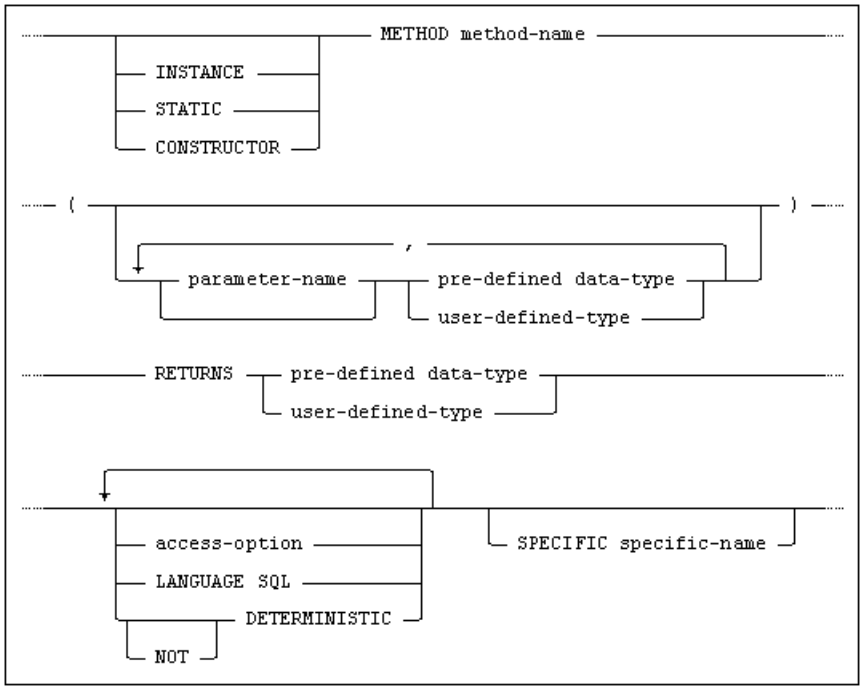
and `type-attributes` is:



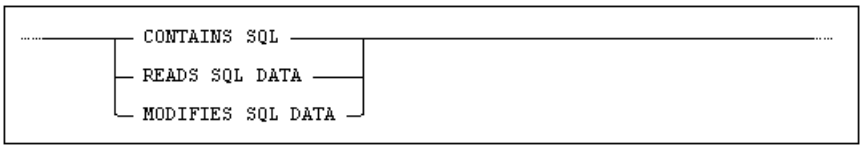
and `cast-option` is:



and `method-specification` is:



where `access-option` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC

Description

A new type is defined. A user-defined type may be used as the data type for columns in `CREATE` or `ALTER TABLE` statements. It can also be used in stored procedures and triggers as the type for variables and parameters.

The *type-name* should follow the normal rules for naming database objects (see *Identifiers* on page *38*). If the *type-name* is unqualified, the type will be created in the schema with the same name as the current ident. If the *type-name* is qualified with a schema name, this schema must be owned by the current ident. The permitted values for data-type are described in *Data Types in SQL Statements* on page *44*.

A distinct type has a single data type whereas a structured type has a list of attributes.

User-defined types are strongly typed, which means that it is only possible to compare values of the same type. When comparing a predefined data type and a distinct user-defined type a type cast must be used. For this purpose there are two routines created automatically when the type is created. Firstly, a function that can be used for casting from the type on which the user-defined type is based to the distinct type. If a cast-source clause is specified the identifier will be used as the name for the function, otherwise the function will have the same name as the user-defined type. Secondly, a function for casting from the user-defined type to the type on which it is based is also created. If cast distinct as source is specified the identifier in this clause is used for the function otherwise the name depends on the source type as seen in the following table.

Source type	Function name
Character	`CHAR`
Character varying	`VARCHAR`
National character	`NCHAR`
National character varying	`NVARCHAR`
Binary	`BINARY`
Binary varying	`VARBINARY`
Integer	`INTEGER`
Decimal	`DECIMAL`
Numeric	`NUMERIC`
Float	`FLOAT`

Source type	Function name
Real	REAL
Double precision	DOUBLE
Date	DATE
Time	TIME
Timestamp	TIMESTAMP
Any interval type	INTERVAL
Boolean	BOOLEAN
Binary large object	BLOB
Character large object	CLOB
National character large object	NCLOB

Examples

```
CREATE TYPE weight AS int;
CREATE FUNCTION checkWeight(w weight) RETURNS boolean RETURN integer(w) > 100;
SET :v = checkWeight(weight(200));

BEGIN
    DECLARE w weight;
    DECLARE i int;
    …
    SET i = integer(w);
    …
    SET w = weight(i);
    …
END
```

Access Options

The following access options may be specified:

- **CONTAINS SQL**

  The method may not contain any data-manipulation-statements. All other procedural-sql-statements are permitted. The method may only invoke methods, functions and procedures with the access option CONTAINS SQL. This option effectively prevents a routine from performing read or write operations on data in the database.

- **READS SQL DATA**

  All procedural-sql-statements are permitted except those performing updates (i.e. DELETE, INSERT and UPDATE). The method may only invoke methods, functions and procedures with the access option CONTAINS SQL or READS SQL DATA.

  This option effectively prevents a routine from performing write operations on data in the database.

- **MODIFIES SQL DATA**

  All procedural-sql-statements are permitted and any method, function or procedure may be invoked from this type of method.

  This option allows a routine to perform read and write operations on data in the database.

If no access options is specified, CONTAINS SQL is implicit

Restrictions

The type-name must be unique within a schema.

A method specification must be unique for a user-defined type with regard to the number of parameters and data types. This means that user-defined type may have multiple method specifications with the same name as long as either the number of parameters differ or if the data types for the parameters differ.

If a parameter name is specified in a parameter list it must be unique within the parameter list.

The ROW data type cannot be used at any place in a type definition.

A domain may not be used as the type for a distinct user-defined type.

The parameter mode for a parameter cannot be specified. It is always IN.

`cast-option` can only be specified for a distinct type.

Notes

When dropping a type with cascade option, any column using that type will be dropped. If this column is the last column in the table, the table will be dropped as well. See DROP TYPE for more details.

The ALTER TYPE statement can be used for adding and dropping method specifications. (See *ALTER TYPE* on page *230*.)

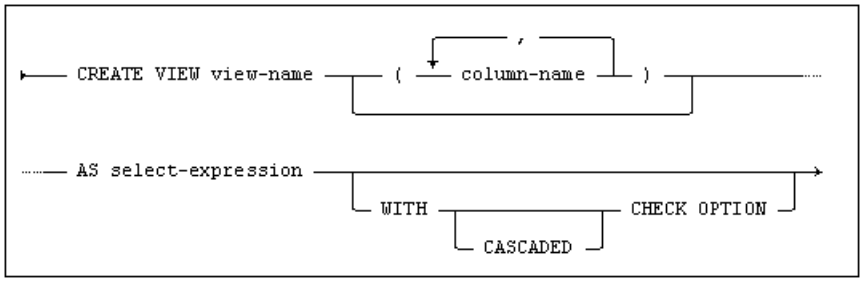
The keywords FINAL and INSTANTIABLE are supported for compliance with SQL-2016. SQL-2016 has support for single inheritance and polymorphism, which is not supported in this version of Mimer SQL.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

CREATE VIEW

Creates a view on one or more tables or views.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

View-name

If `view-name` is specified in its unqualified form, the view will be created in the schema which has the same name as the current ident.

If `view-name` is specified in its fully qualified form (i.e. `schema-name.view-name`) the view will be created in the named schema (in this case, the current ident must be the creator of the specified schema).

Column-name

If a list of column names is given in parentheses before the `query-expression`, the columns in the view are named in accordance with this list. There must be the same number of names in the column list as there are columns addressed by the `query-expression`. The names must be unique within the list.

If the column name list is omitted, the columns in the view will be given the same names as they have in the source table(s) or view(s) addressed in the select-specification. The column names in the source must all be unique in the view being created. If this is not the case, an explicit column name list must be given. An explicit column name list must also be given if columns in the view are defined as expressions without correlation names.

select-expression

A view is created in accordance with the specification in the `query-expression`, see *SELECT* on page *398* for more information on `select-expression`'s.

WITH CHECK OPTION

Specification of `WITH CHECK OPTION` indicates that any data inserted into the view by `INSERT` or `UPDATE` statements will be checked for conformity with the definition of the view. Attempts to insert data which do not conform to the view definition will be rejected.

The optional keyword `CASCADED` can be explicitly specified in the `WITH CHECK OPTION` clause to ensure that any data inserted into a view which is based on this view will be also be checked for conformity with the definition of this view.

Thus, if an INSERT or UPDATE in a view based on this one results in an attempt to insert data into this view which does conform to the view definition, the data change operation will be rejected.

If CASCADED is not specified, it is assumed by default (use of the keyword CASCADED is now permitted to allow for future extensions to the Mimer SQL syntax).

Language Elements

query-expression, see *SELECT* on page *398*.

Restrictions

CREATE VIEW requires SELECT access to the tables or views from which the view is created, and EXECUTE privilege on routines and USAGE privilege on sequences and domains referenced.

Notes

The view name may not be the same as the name of any other table, view, index, constraint or synonym belonging to the same schema.

The creator of the view is always granted SELECT access to the view. If the view is updatable, see below, any access the creator may hold on the underlying table or view at the time the new view is created is also granted on the new view. Access to the view is granted WITH GRANT OPTION only if the corresponding access to all underlying tables, views, routines, sequences and domains are held WITH GRANT OPTION.

SELECT and UPDATE statements can only be performed on data accessible from the view. Insertion of a new row assigns the default value or null value to columns in the base table excluded from the view, in accordance with the definition of the columns. Deletion of a row from a view removes the entire row from the underlying base table, including columns invisible from the view.

The select-specification defining the view may not contain references to host variables.

The WITH CHECK OPTION clause is illegal if the view is not updatable. A result set is only updatable if all of the following conditions are true:

- the keyword DISTINCT is not specified
- the FROM clause specifies exactly one table reference and that table reference refers either to a base table or an updatable view
- a GROUP BY clause is not included
- a HAVING clause is not included.
- the result set is not the product of an explicit INNER or OUTER JOIN
- the keyword UNION is not included
- the keyword EXCEPT is not included
- the keyword INTERSECTION is not included

A view will always be updatable if an INSTEAD OF trigger exists on the view, regardless of the conditions previously mentioned. If there is an INSTEAD OF trigger any possible with check option for the view is ignored. If all the INSTEAD OF triggers on the view are dropped, the view will revert to not updatable if one or more of those conditions are not true.

If an updatable view is based on other views, insert and update operations are checked against all view definitions for which WITH CHECK OPTION is specified. Thus if view-2 is defined with check option on view-1, which in turn is defined with check option on a base table, no changes may be made in the base table through either view-1 or view-2 which violate the definition of view-1.

Example

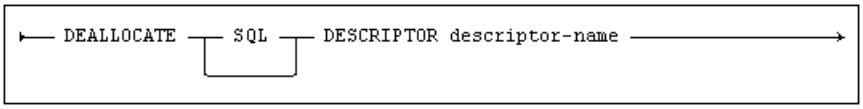
```
CREATE VIEW mimer_store_book.details
   AS SELECT authors_list, product AS title, producer AS publisher, format,
             price, stock, reorder_level,
             extract_date(release_date) AS release_date,
             'ISBN:99-999-9999-9' as ISBN,  -- *****
             -- *****'ISBN:' || mimer_store_book.format_isbn(isbn) AS isbn,
             ean_code, status, product_search AS title_search,
             product_details.item_id, category_id, product_id,
             display_order, image_id
   FROM product_details
     JOIN mimer_store_book.titles
       ON product_details.item_id = mimer_store_book.titles.item_id;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

DEALLOCATE DESCRIPTOR

Deallocates an SQL descriptor area.



Usage

Embedded, Module.

Description

This statement deallocates an SQL descriptor area that was previously allocated with the specified `descriptor-name`.

Notes

An SQL descriptor area with the specified name must have been previously allocated.

Example

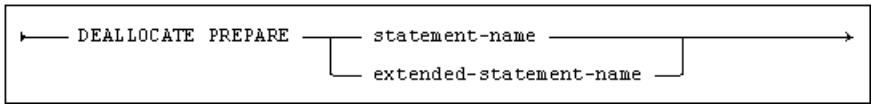
```
EXEC SQL DEALLOCATE DESCRIPTOR 'SDA';
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, “Basic dynamic SQL”. Feature B032, “Extended dynamic SQL” support for dynamic descriptor names.

DEALLOCATE PREPARE

Deallocates a prepared SQL statement.



Usage

Embedded, Module.

Description

The prepared statement associated with the statement name is destroyed. Any cursor allocated with an `ALLOCATE CURSOR` statement that is associated with the prepared statement is also destroyed.

See *ALLOCATE CURSOR* on page *196* for a description of extended statements.

Notes

The statement name must identify a statement prepared in the same compilation unit.

The statement must not identify an existing prepared statement that is associated with an open cursor.

Example

```
EXEC SQL DEALLOCATE PREPARE :stm1;
```

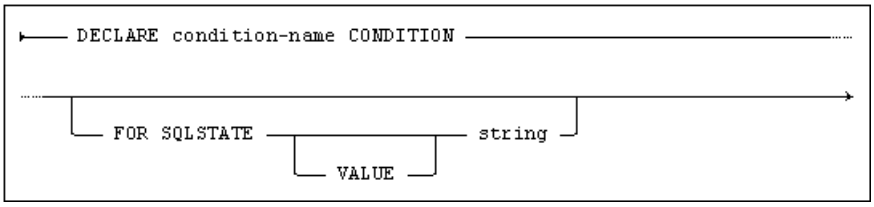
For more information, see the *Mimer SQL Programmer's Manual, Chapter 4, Dynamic SQL*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B032, "Extended dynamic SQL".

DECLARE CONDITION

Declares a condition name for an exception condition value.



Usage

Procedural.

Description

A condition declaration defines an identifier that can be used as a name for exceptions and/or `SQLSTATE` values. The identifier can be used in `SIGNAL` and `RESIGNAL` statements as well as in handler declarations. This results in a code that is easier to read and understand compared with using explicit `SQLSTATE` values.

A condition identifier can optionally be associated with an `SQLSTATE` value. Using a condition identifier with an associated `SQLSTATE` value in a `SIGNAL` or `RESIGNAL` statement means that the associated `SQLSTATE` value is signaled as well. If the condition identifier does not have an associated `SQLSTATE` value, the `SQLSTATE` value `45000` is signaled.

Restrictions

A condition name which represents an `SQLSTATE` value may only be declared to represent a specific `SQLSTATE` value, i.e. it is not possible to declare a condition name to represent an exception class group. For a description of exception class groups, see *DECLARE HANDLER* on page *312*.

The scope of a condition name covers all the procedural SQL statements in the compound statement declaring it, including any other compound statements nested within it.

The general naming rules for a condition name are the same as those for other database objects.

If a condition name is declared to represent a particular `SQLSTATE` value, another condition name cannot exist for that same `SQLSTATE` value which has exactly the same scope.

A condition name cannot be declared for an `SQLSTATE` value with class ‘successful completion’, this covers all `SQLSTATE` values starting with `00`.

Notes

The `SQLSTATE` value string is five characters long and contains only alphanumeric characters.

In Mimer SQL any `SQLSTATE` value that falls outside the range of standard `SQLSTATE` values is treated as an implementation-defined value.

Standard SQLSTATE values begin with the characters A-I, S, 0-4 and 7, while implementation-defined SQLSTATE values begin with the characters J-R, T-Z, 5-6 and 8-9.

Example

```
DECLARE invalid_parameter CONDITION;

DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN
    DECLARE condition_name VARCHAR(128);
    GET DIAGNOSTICS EXCEPTION 1 condition_name = CONDITION_IDENTIFIER;
    IF condition_name = 'invalid_parameter' THEN
       ...
    END IF;
END;

SIGNAL invalid_parameter;
```

For more information, see *Appendix E Return Status and Conditions* or *Mimer SQL Programmer's Manual, Chapter 11, Declaring Condition Names*.

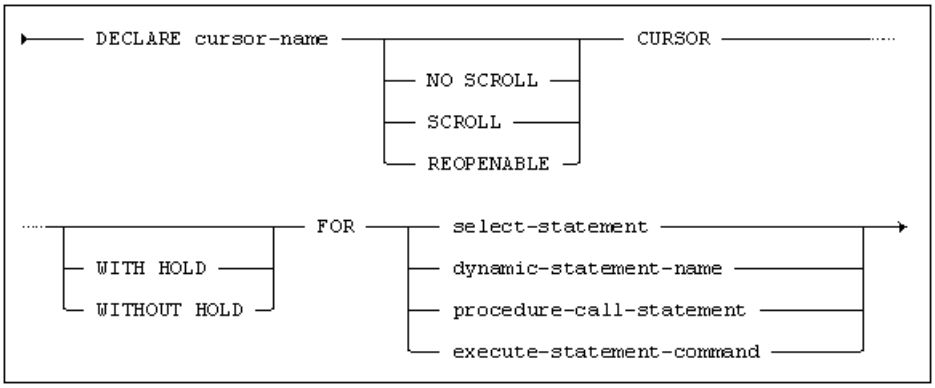
For information about the SIGNAL statement, see *SIGNAL on page 422*. For information about the RESIGNAL statement, see *RESIGNAL on page 384*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".

DECLARE CURSOR

Declares a cursor definition.



Usage

Embedded, Module, Procedural.

Description

A cursor is declared in accordance with the `select-statement` or the result set procedure call specified in `procedure-call-statement`.

The `select-statement` may be specified explicitly in ordinary embedded SQL applications or by the name of a prepared `SELECT`, identified by `dynamic-statement-name`, in dynamic SQL statements, see the *Mimer SQL Programmer's Manual, Chapter 4, Dynamic SQL*.

The cursor is identified by `cursor-name`, and may be used in `FETCH`, `DELETE CURRENT` and `UPDATE CURRENT` statements. The cursor must be activated with an `OPEN` statement before it can be used.

A cursor declared as `REOPENABLE` may be opened several times in succession, and previous cursor states are saved on a stack, see *OPEN* on page *378*. Saved cursor states are restored when the current state is closed, see *CLOSE* on page *237*.

A cursor declared as `SCROLL` will be a scrollable cursor. For a scrollable cursor, records can be fetched using an orientation specification. See the description of *FETCH* on page *339* for a description of how the orientation can be specified.

A cursor declared `WITH HOLD` will be a holdable cursor. Open holdable cursors are not closed when a transaction is committed. A cursor will be non-holdable if `WITHOUT HOLD` is explicitly specified.

`WITHOUT HOLD` and `NO SCROLL` are default cursor attributes and do not have to be specified.

Language Elements

`select-statement`, see *SELECT Statements* on page *398*.

`procedure-call-statement`, see *CALL* on page *233*.

Restrictions

A cursor for a result set procedure call must not be declared WITH HOLD.

If an execute-statement-command is used, the precompiled statement must be a SELECT or a result set procedure CALL.

If a procedure-call-statement is specified, it must specify a result set procedure.

The following restrictions apply to procedural usage:

- The cursor cannot be declared as REOPENABLE
- The SELECT statement cannot be in the form of a prepared dynamic SQL statement, i.e. specifying dynamic-statement-name is not allowed
- If the cursor declaration contains a SELECT statement, the access-clause for the procedure must be READS SQL DATA or MODIFIES SQL DATA, see *CREATE PROCEDURE* on page *271*
- The execute-statement-command is not allowed.

Notes

The DECLARE CURSOR statement is declarative, not executable. In an embedded usage context, access rights for the current ident are checked when the cursor is opened, not when it is declared.

In a procedural usage context, access rights for the current ident are checked when the cursor is declared, i.e. when the procedure containing the declaration is created.

The value of cursor-name may not be the same as the name of any other cursor declared within the same compound statement (Procedural usage) or in the same compilation unit (Embedded usage).

The select-statement is evaluated when the cursor is opened, not when it is declared. This applies both to select-statement's identified by statement name, and to host variable references used anywhere in the SELECT statement.

The execution of the result set procedure specified in a CALL statement is controlled by the opening of the cursor and subsequent fetches, see the *Mimer SQL Programmer's Manual, Chapter 11, Result Set Procedures*.

REOPENABLE cannot be used if evaluation of select-statement uses a work table, or if the cursor declaration occurs within a procedure.

If the declared cursor is a dynamic cursor, the DECLARE statement must be placed before the PREPARE statement.

Cursors should normally be declared WITHOUT HOLD (default), because WITH HOLD cursors require more internal resources then ordinary cursors.

A reopenable cursor can be used to solve the 'Parts explosion' problem. Refer to the *Mimer SQL Programmer's Manual, Chapter 4, The 'Parts explosion' Problem* for a description.

Examples

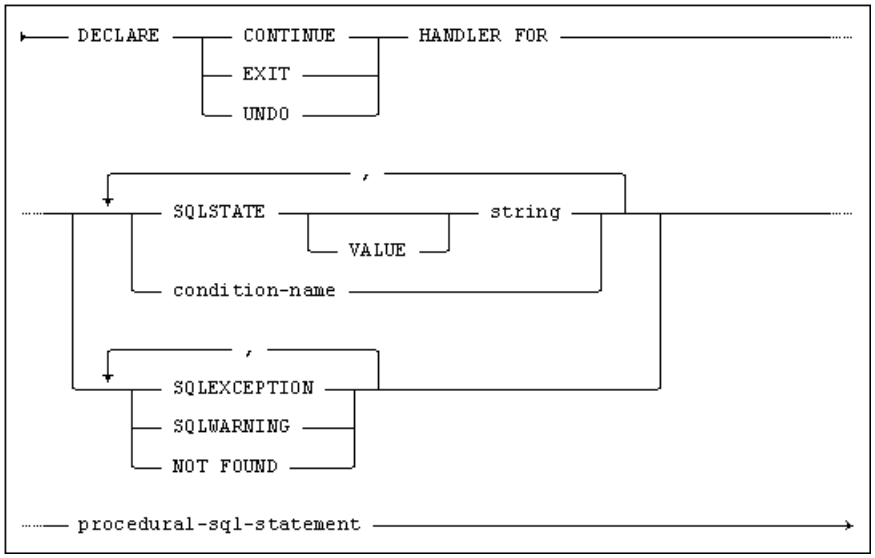
```
DECLARE cur2 SCROLL CURSOR FOR SELECT c1,c2 FROM tab1;

DECLARE cur3 CURSOR WITH HOLD FOR stmt1;

DECLARE cur1 CURSOR FOR EXECUTE STATEMENT seltaba;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F431, “Read-only scrollable cursor”, support for the `SCROLL` keyword. Feature F831, “Full cursor update” Feature T551, “Optional keywords for default syntax” support for the WITHOUT HOLD clause.
	Mimer SQL extension	The keyword REOPENABLE is a Mimer SQL extension. Support for EXECUTE STATEMENT and CALL statement in a cursor definition is a Mimer SQL extension.

DECLARE HANDLER

Declares an exception handler.



Usage

Procedural.

Description

An exception handler may be declared to respond to a specific exception condition or one or more of the exception class groups `SQLEXCEPTION`, `SQLWARNING` or `NOT FOUND`. An exception handler that responds to one or more exception class groups is called a general exception handler.

A specific exception condition may be specified by using an `SQLSTATE` value or a condition name, see *DECLARE CONDITION* on page *307*. An exception handler that responds to one or more specific exception conditions is called a specific exception handler.

The keywords `CONTINUE`, `EXIT` and `UNDO` affect the flow of control behavior subsequent to the execution of the exception handler.

If `CONTINUE` is specified, the flow of control continues by executing the SQL statement immediately following the statement that raised the error, after the handler has executed.

If `EXIT` is specified, the flow of control exits the compound statement within which the exception handler is declared after the handler has executed.

If `UNDO` is specified, all the changes made by the SQL statements in the `ATOMIC` compound statement, within which the handler is declared, (or by any SQL statements triggered by those changes) are canceled. Then the handler is executed and the flow of control exits the compound statement.

Restrictions

An UNDO exception handler can only be declared within an ATOMIC compound statement.

An exception handler must be either a general or a specific exception handler, it cannot respond to both an exception class group and a specific exception condition.

The same exception condition must not be specified more than once, whether by SQLSTATE value or by condition name, in an exception handler declaration.

Within a given scope, only one specific exception handler may be declared for a particular exception condition.

If string is specified, it must have length five and contain only alphanumeric characters.

Notes

Within a given compound statement, if both a general and a specific exception handler have been declared to respond to a given exception condition, the specific exception handler will handle the exception.

If no exception handlers have been declared to respond to an exception condition in the compound statement within which the error was raised, the exception condition is propagated out to the enclosing compound statement or to the calling environment.

The above does not apply to exception conditions with the NOT FOUND and SQLWARNING class, these are not propagated by the exception handling mechanism in absence of an exception handler defined to handle them.

SQLWARNING covers SQLSTATE values beginning with 01.

NOT FOUND covers SQLSTATE values beginning with 02.

SQLEXCEPTION covers all other SQLSTATE values (including implementation-defined values), except those beginning with 00.

Example

```
S1:
BEGIN
   DECLARE EXIT HANDLER FOR SQLEXCEPTION
   BEGIN
      ...
      ...
   END;
   ...
   ...
 END S1;
```

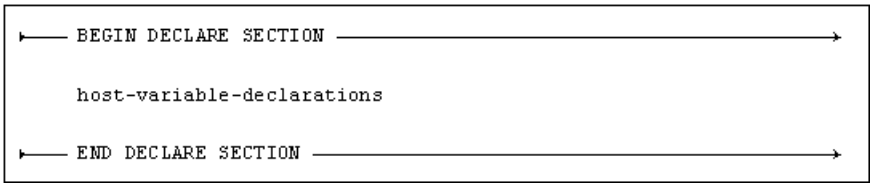
For more information, see *Appendix E Return Status and Conditions* or the *Mimer SQL Programmer's Manual, Chapter 11, Declaring Exception Handlers*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".

DECLARE SECTION

Identifies declaration of host variables used in SQL statements.



Usage

Embedded.

Description

Host variables declared between the `BEGIN DECLARE SECTION` and `END DECLARE SECTION` statements are declared for both the host language compiler and the SQL compiler. Host variables used in SQL statements must be declared in an `SQL DECLARE SECTION`.

Notes

Host variables declared in the `SQL DECLARE SECTION` must be elementary data items or pointers to character strings. They may not be declared as part of a record or structure.

The rules for placing the `DECLARE SECTION` in the host program code and for declaring variables within the `SECTION` follow the host language syntax.

Language-specific issues are discussed in the *Mimer SQL Programmer's Manual, Appendix A, Host Language Dependent Aspects*.

SQL statements may not be placed within the `SQL DECLARE SECTION`.

`BEGIN DECLARE SECTION` and `END DECLARE SECTION` are two separate SQL statements, each of which must be preceded by `EXEC SQL` and followed by the (language-specific) statement terminator.

Example

The following example, in C, shows the correct usage of both statements:

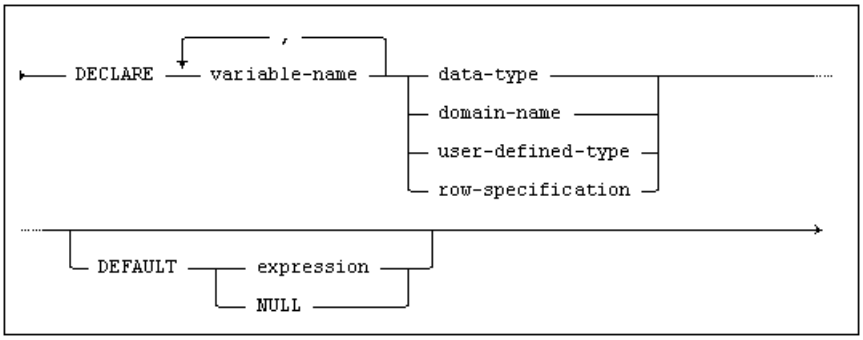
```
exec sql BEGIN DECLARE SECTION;
    char SQLSTATE[6];
    char usernm[129], passwd[129];
    NVARCHAR tablenm[129];
exec sql END DECLARE SECTION;
```

Standard Compliance

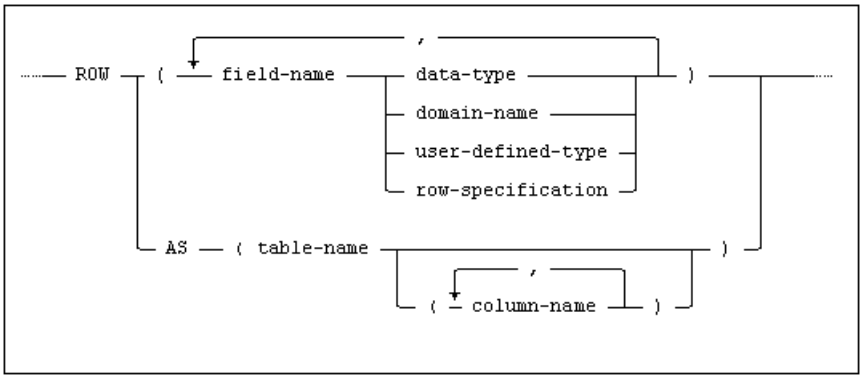
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

DECLARE VARIABLE

Declares a variable.



where `row-specification` is:



Usage

Procedural.

Description

The value for data-type can be any data type supported by Mimer SQL, see *Data Types in SQL Statements* on page *44*.

More than one variable of the same type can be declared in a single declaration.

It is possible to declare a variable as a record, by using the row specification.

The fields in a record may themselves be records, to an unlimited depth. To reference a field in a record the notation `recordVariable.fieldName` is used.

A record can be declared as being the same as a table or a part of table with the `AS` clause. This means that the fields in the record will have the same name and type as the columns in the table. If the column list is omitted, all columns are used.

The optional `DEFAULT` clause may be used to specify an initial value for the variable(s). A value of null is permitted as the value for the `DEFAULT` clause.

If a `ROW` data type definition has been specified for data-type, a row value expression can be specified for expression in the `DEFAULT` clause.

If the `DEFAULT` clause is not specified, the variable(s) will be set to null initially.

In the case of a variable declared with the `ROW` data type, each field in the variable is set to null initially if a `DEFAULT` clause is not specified.

Restrictions

The name of a variable cannot be the same as any of the routine parameter names.

A function with `MODIFIES SQL DATA` specified for its access clause cannot be used as `expression` in the `DEFAULT` clause.

Notes

It is possible to declare a variable with the same name as that of a column in a table. In such a situation, an unqualified name will always resolve to the table column name and not the variable. We recommend that a suitable naming convention be adhered to that distinguishes between the two.

If a variable is defined as using a domain, any assignment to this parameter will be verified to ensue that any check constraint is not violated. If the domain has a default value, the variable will be initialized with this value unless there is an explicit default clause in the declaration.

Examples

```
DECLARE orderNumber INTEGER DEFAULT 0;

DECLARE firstName,lastName VARCHAR(30);

DECLARE purchase row(customerId integer, orderNumber integer,
                     purchaseDate date, productId integer, quantity integer)
   DEFAULT (0,0,current_date,0,0);

DECLARE book ROW AS (mimer_store_book.details);

DECLARE bookTitle ROW AS(mimer_store_book.details(isbn,title));
```

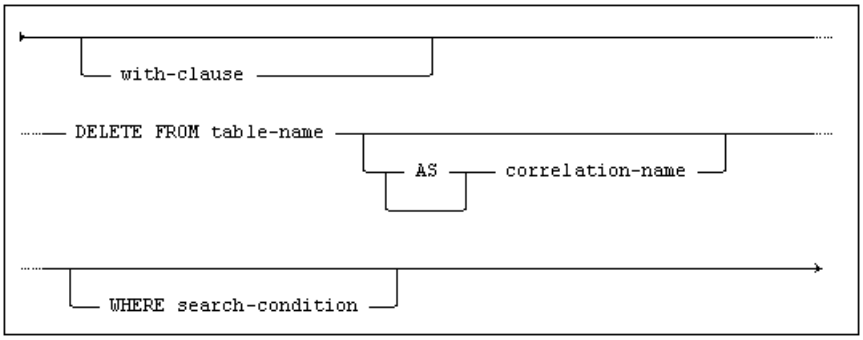
For more information, see *Mimer SQL Programmer's Manual, Chapter 11, Declaring Variables*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".
	Mimer SQL extension	The use of AS-clause is a Mimer SQL extension.
	Mimer SQL extension	The possibility to use domains in PSM is a Mimer SQL extension.

DELETE

Deletes a set of rows from a table or view.



Usage

Embedded, Interactive, Module, ODBC, Procedural, JDBC.

Description

All rows in the set defined by the `WHERE` clause are deleted from the base table or view identified by `table-name`. If no `WHERE` clause is specified, all rows are deleted.

If `table-name` identifies a view rather than a base table, entire rows, including columns invisible from the view, are deleted from the base table on which the view is defined. For a delete to be performed on a view, the view must be updatable, see *CREATE VIEW* on page *302*.

A `NOT FOUND` condition code is returned if no row is deleted, see *Appendix E Return Status and Conditions*.

Language Elements

`search-condition`, see *Search Conditions* on page *165*.

`with-clause`, see *The WITH Clause* on page *179*.

Restrictions

`DELETE` privilege must be held on the table or view identified by `table-name`.

In a procedural usage context, the `DELETE` statement is only permitted if the procedure `access-clause` is `MODIFIES SQL DATA`, see *CREATE PROCEDURE* on page *271*.

Notes

If a `correlation-name` is specified after the table name in the `DELETE FROM` clause, the correlation name must be used to refer to the table in the `WHERE` clause of the `DELETE` statement.

If the table name addressed by the DELETE statement is subject to any referential constraints, the delete operation must not create a situation where these constraints are violated. The effect of the delete operation on any referential constraints depends on the delete-rule in effect for each constraint, see *CREATE TABLE* on page *285* for a description of the delete-rule options.

A DELETE statement is executed as a single statement. If an error occurs at any point during the execution, none of the rows in the set defined by the WHERE clause will be deleted (however, if the table is stored in a databank with the WORK option it is possible that some rows will be deleted).

Example

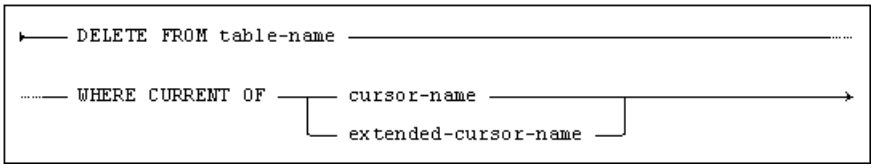
```
DELETE FROM countries
    WHERE CITY = 'Dublin';
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

DELETE CURRENT

Deletes the current row indicated by a cursor.



Usage

Embedded, Module, ODBC, Procedural, JDBC.

Description

The row currently indicated by the cursor is deleted from the base table implied by the cursor definition.

The row indicated by the cursor is that retrieved by the most recent `FETCH` statement, see *FETCH* on page *339*.

After the deletion, the cursor is no longer positioned on a row in the set defined by the cursor declaration. A `FETCH` statement is required to position the cursor on the next row in the set before another `DELETE CURRENT` or an `UPDATE CURRENT` statement can be executed.

See *ALLOCATE CURSOR* on page *196* for a description of extended cursors.

If an `extended-cursor-name` is specified in a `DELETE CURRENT` statement, it specifies the name of a host variable whose value is the name of the cursor.

Restrictions

`DELETE` access to the table or view identified by `table-name` is required when the cursor used for the `DELETE CURRENT` statement is opened. If `DELETE` access is not held, the cursor may be opened but `DELETE CURRENT` statements will fail.

A row indicated by a `WITH HOLD` cursor must have been fetched in the same transaction.

The `DELETE CURRENT` statement may not be used for read-only cursors.

A cursor cannot be identified by specifying `extended-cursor-name` in a procedural usage context.

In a procedural usage context, the `DELETE CURRENT` statement is only permitted if the procedure `access-clause` is `MODIFIES SQL DATA`, see *CREATE PROCEDURE* on page *271*.

Notes

The table name given in the `DELETE CURRENT` statement must be the same as that specified in the `FROM` clause of the cursor declaration. If a synonym is used in one of the statements, the same synonym must also be used in the other.

If the table name addressed by the `DELETE CURRENT` statement is subject to any referential constraints, the delete operation must not create a situation where these constraints are violated. The effect of the delete operation on any referential constraints depends on the `delete-rule` in effect for each constraint, see *CREATE TABLE* on page *285* for a description of the `delete-rule` options.

If an error occurs during execution of a `DELETE CURRENT` statement, the row is not deleted.

Example

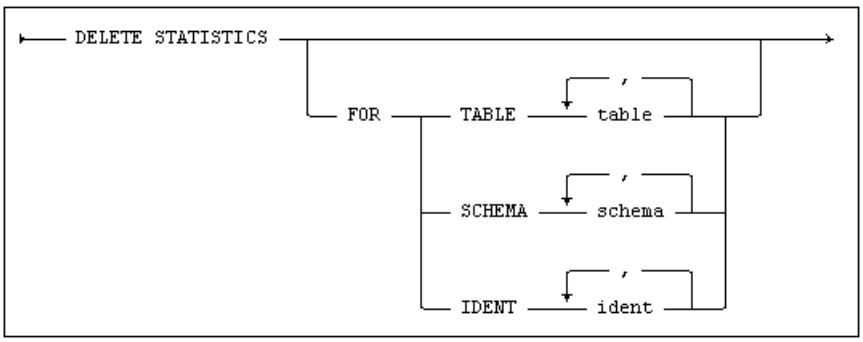
```
exec sql FETCH cur1 INTO :ival;
if (ival < 0) {
    exec sql DELETE FROM tab1 WHERE CURRENT OF cur1;
    …
}
else…
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

DELETE STATISTICS

Deletes the statistics recorded for all tables in the database, a specified list of tables, all tables in a specified list of schemas or all the tables belonging to the schemas owned by a specified list of idents.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The default operation is to delete statistics for all tables, including data dictionary tables, in the database.

It is possible to delete statistics for a specified list of tables by using the `FOR TABLE` option, for all tables belonging to a specified list of schemas by using the `FOR SCHEMA` option or for all the tables belonging to the schemas created by a specified list of idents by using the `FOR IDENT` option.

Restrictions

The current ident must be the creator of all the tables involved or must have `STATISTICS` privilege.

Notes

The `DELETE STATISTICS` statement can be used concurrently with other SQL statements.

Example

```
DELETE STATISTICS FOR IDENT joe;
```

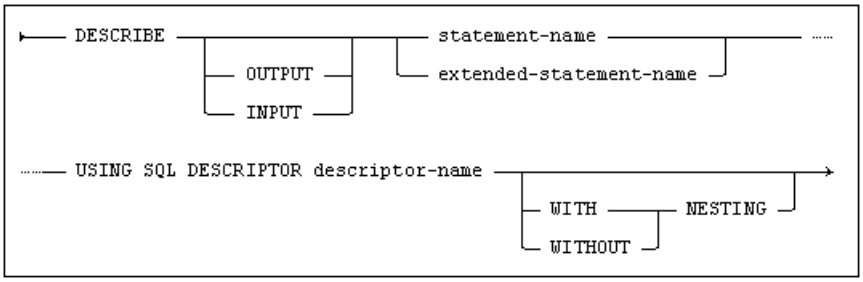
For more information, see the *Mimer SQL System Management Handbook, Chapter 11, Database Statistics*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The DELETE STATISTICS statement is a Mimer SQL extension.

DESCRIBE

`DESCRIBE` gathers information on the result set or dynamic variable specifications in a prepared statement and places the information in an SQL descriptor area.



Usage

Embedded, Module.

Description

Information concerning the prepared statement identified by `statement-name` is placed in the named SQL descriptor area.

The `DESCRIBE` statement and the `DESCRIBE OUTPUT` statement are equivalent, and gather information on the items in the result set of the statement and places it in the SQL descriptor area.

The `DESCRIBE INPUT` statement describes the statement's input and output variables.

The `DESCRIBE OUTPUT` statement describes the result set returned by the statement (a `SELECT` or a `CALL` to a result set procedure).

If descriptions of both the result set and the input variables are required for a prepared statement, the statement must be described separately with each form of `DESCRIBE`.

The `descriptor-name` is identified by a host variable or a literal.

`WITHOUT NESTING` is implicit.

See *ALLOCATE DESCRIPTOR* on page *198* for a description of the SQL descriptor area.

See *ALLOCATE CURSOR* on page *196* for a description of extended statements.

See also, *GET DESCRIPTOR* on page *344*.

Restrictions

The `DESCRIBE` statement is only applicable to dynamically prepared SQL statements.

Examples

```
exec sql ALLOCATE DESCRIPTOR 'DescrIn';
exec sql ALLOCATE DESCRIPTOR 'DescrOut';
sqlstr = 'SELECT * FROM tab1 WHERE col1 = ? and col2 = ?';
exec sql PREPARE 'stmt X1' FROM :sqlstr;
exec sql DESCRIBE INPUT 'stmt X1' USING SQL DESCRIPTOR 'DescrIn';
exec sql DESCRIBE OUTPUT 'stmt X1' USING SQL DESCRIPTOR 'DescrOut';

sqlstr = 'CALL proc(?, ?, ?)';
exec sql PREPARE stmt2 FROM :sqlstr;
exec sql DESCRIBE INPUT stmt2 USING SQL DESCRIPTOR :SQLA1;
```

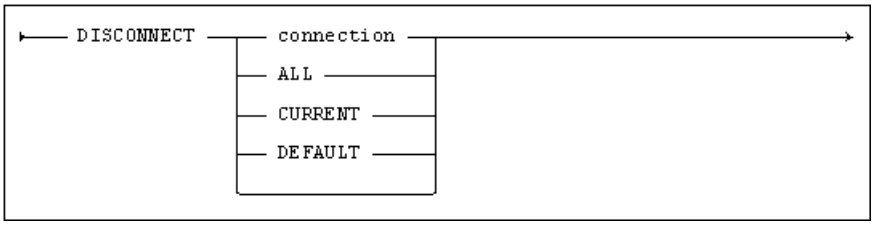
For more information, see the *Mimer SQL Programmer's Manual, Chapter 4, Describing Prepared Statements*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, "Basic dynamic SQL".

DISCONNECT

Disconnects a user from the specified connection.



Usage

Embedded, Interactive, Module.

Description

The specified connection is disconnected. Any current transaction is rolled back, cursors are closed, and compiled statements are destroyed. No further access to the database using that connection is possible.

If the specified connection is not the current connection the application still has access to the current connection. Otherwise, to continue with another connection the application must either perform a `SET CONNECTION` or a `CONNECT` statement.

The `connection` is not case-sensitive and may be specified as a literal value or by using a host variable.

If no disconnect option is specified, `CURRENT` is assumed by default.

The `DISCONNECT` statement may not be issued within a transaction. The transaction must first be ended using `COMMIT` or `ROLLBACK`, before `DISCONNECT` can be performed.

Example

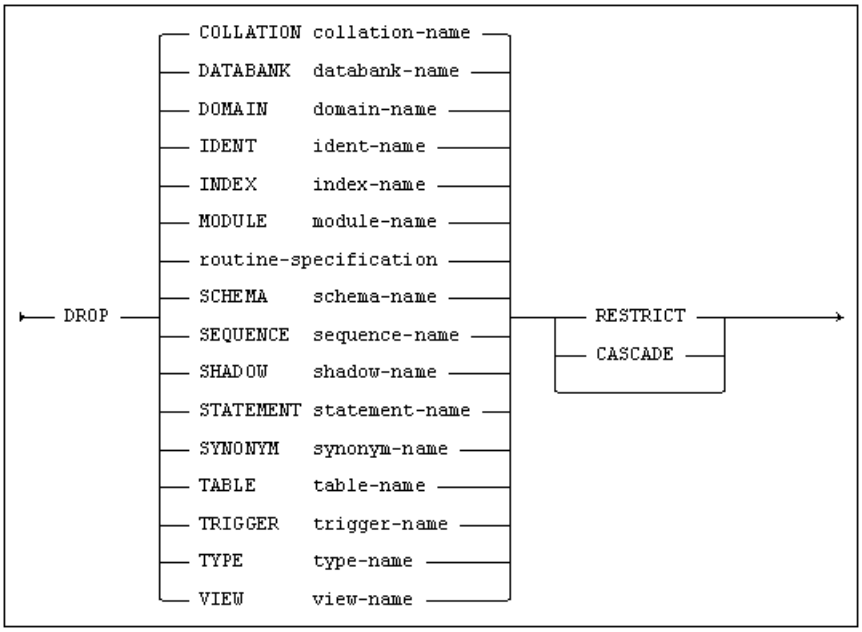
```
DISCONNECT 'connection 1';
```

Standard Compliance

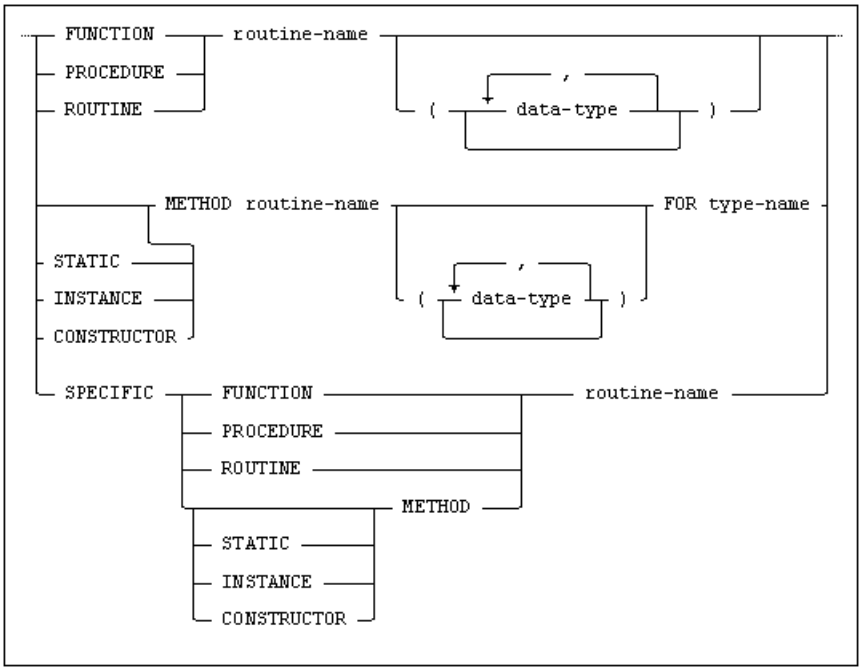
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F771, “Connection management”.
	Mimer SQL extension	Support for DISCONNECT without any option is a Mimer SQL extension.

DROP

Drops an object from the database.



where `routine-specification` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The named object is deleted from the database. The object name is free to be reused for other objects.

The CASCADE and RESTRICT keywords specify the action to be taken if other objects exist that are dependent on the object being dropped. If CASCADE is specified, such objects will be dropped as well. If RESTRICT is specified, an error is returned if other objects are affected, and no objects are dropped.

If neither RESTRICT nor CASCADE is specified, then RESTRICT is implicit.

Restrictions

A database object can only be dropped by its creator, unless it is implicitly dropped because of cascade effects when another object is dropped, see the Notes section below.

You must have exclusive use of a table to drop the table or an index on the table, and of a databank to drop the databank.

DROP SHADOW is only for use with the optional Mimer SQL Shadowing module, and requires SHADOW privilege.

The databank for which the shadow exists cannot be used by any other user while the shadow is being dropped.

Only the creator of a STATEMENT can drop it. Neither RESTRICT nor CASCADE is supported, since DROP STATEMENT will never cause any cascading effects to occur.

Notes

DROP COLLATION

You can drop a collation only if there are no dependencies.

DROP COMMENT

Comments may not be dropped from the data dictionary, but they may be replaced by blank comments, see *COMMENT* on page *239*.

DROP DATABANK

When a databank is dropped, all tables and sequences in the databank are dropped. All shadows defined on the databank are also dropped. An attempt is made to delete the physical file in which the databank is stored. If the file deletion is unsuccessful for any reason (e.g. the disk is not mounted), the databank is dropped from the database but the file remains.

If the databank is OFFLINE, no attempt is made to delete the physical databank file or any shadow file(s).

DROP DOMAIN

When a domain is dropped, existing columns defined using the domain retain all the properties of the domain. No new columns may however use the domain. All routines, triggers or views whose definitions contain a CAST involving the domain will be dropped.

DROP FUNCTION

When a function is dropped with the CASCADE option in effect, all constraints, functions, procedures, triggers or views invoking it will be dropped. Dropping any object referenced from the SQL statements in the body of a function will drop the function when the CASCADE option is in effect.

DROP IDENT

When an ident is dropped, all objects owned by the ident are dropped, and all privileges granted by the ident are revoked. (Remember that revocation of privileges, in particular, may have recursive effects on other objects.)

DROP INDEX

When an index is dropped and CASCADE is in effect, all the objects (i.e. functions, procedures, statements, triggers and views) explicitly referencing the index are also dropped.

DROP MODULE

When a module is dropped, all the routines belonging to the module are also dropped.

DROP PROCEDURE

When a procedure is dropped with the CASCADE option in effect, all other routines or triggers calling it will be dropped. Dropping any object referenced from the SQL statements in the body of a procedure will drop the procedure when the CASCADE option is in effect.

DROP SCHEMA

When a schema is dropped and CASCADE is in effect, all the objects belonging to the schema are also dropped. If RESTRICT is in effect, the schema will be dropped only if it is empty.

DROP SEQUENCE

When a sequence is dropped and CASCADE is in effect, all the objects (i.e. domains, functions, procedures, table columns, triggers and views) referencing the sequence are also dropped.

DROP SHADOW

DROP SHADOW deletes the named shadow from the data dictionary.

An attempt is made to delete the physical shadow file in the same way as for dropping a databank. If the shadow or the master databank is OFFLINE however, no attempt is made to delete the physical shadow file.

DROP STATEMENT

A statement may not be dropped when it is in use.

DROP SYNONYM

There are no cascade effects when a synonym is dropped because it is resolved to the associated table or view when an SQL statement containing the synonym is executed. Thus, it is a table or view reference that is actually stored in the database, not the synonym reference. Once dropped, of course, the synonym can no longer be used in new SQL statements.

DROP TABLE

When a table is dropped, all views based on that table and all triggers created on it are also dropped.

When a table referenced from within a routine, trigger or statement is dropped with the `CASCADE` option in effect, the routine, trigger or statement will also be dropped, see also the notes above for Function, Module, Procedure and Trigger for full cascade implications.

If a table used as a `REFERENCES` table in a `FOREIGN KEY` clause is dropped, the referential integrity constraint is lost from the table with the foreign key clause.

All cursors defined for a table must be closed before the table can be dropped.

DROP TRIGGER

If a trigger has been created on a non-updatable view, the creator of the trigger implicitly gets the appropriate privilege for the trigger event on that view, with `WITH GRANT OPTION`.

The creator of the trigger may then have granted the privilege to other idents or may have used the privilege to perform updates on the view in one or more routines subsequently created.

If the trigger is then dropped, with the `CASCADE` option in effect, any routines using the privilege to update the view will be dropped and the privilege will be revoked from any idents to whom the trigger creator granted it.

DROP VIEW

When a view is dropped, all other views based on that view and all triggers created on it are also dropped.

When a view referenced from within a routine, trigger or statement is dropped with the `CASCADE` option in effect, the routine, trigger or statement will also be dropped, see also the notes above for Function, Procedure and Module for full cascade implications.

Example

```
DROP IDENT joe CASCADE;
```

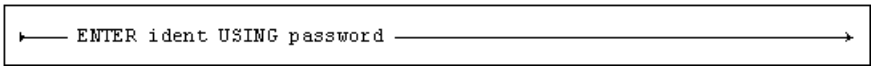
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F032, “CASCADE drop behavior” support for the cascade option. Feature F251, “Domain support” support for drop domain statement. Feature F690, “Collation support” support for drop collation statement. Feature T211, “Basic trigger capability” support for drop trigger statement.
	Mimer SQL extension	DROP DATABANK, DROP IDENT, DROP INDEX, DROP STATEMENT, DROP SHADOW and DROP SYNONYM are Mimer SQL extensions. Optional CASCADE or RESTRICT is a Mimer SQL extension.

ENTER

Connects a PROGRAM ident to the system.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The specified ident becomes the current ident, provided that the password is correct.

The ident and the password can be specified either as a host variable or as a literal.

A PROGRAM ident is set up to have certain privileges and access to the database which apply after the ENTER statement has been used and replace those that apply to the ident executing the ENTER statement.

Restrictions

The ENTER statement may only be issued for idents of type PROGRAM.

ENTER may not be used to log on to the system.

To perform an ENTER operation for a PROGRAM ident, the current ident must have EXECUTE privilege on that ident.

The ENTER statement may not be issued inside a transaction.

Notes

When the ENTER operation is successfully performed, the privileges granted to the new current ident apply and those granted to the previous current ident are suspended until the PROGRAM ident is left, see *LEAVE (PROGRAM ident)* on page *375*.

Example

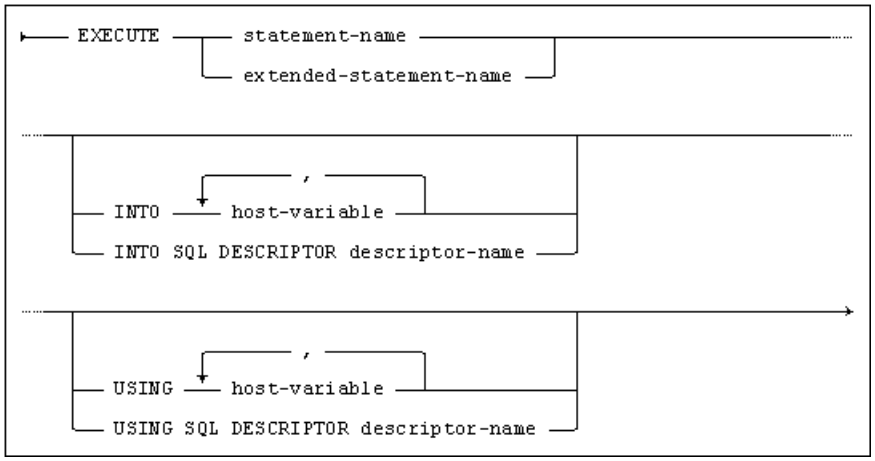
```
ENTER 'TSTPRG' USING 'secret';
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	Support for the ENTER statement is a Mimer SQL extension.

EXECUTE

Associates parameter values with a previously prepared SQL statement and executes the statement.



Usage

Embedded, Module.

Description

The prepared statement identified by the statement name is executed.

If the source form of the prepared statement contains parameter markers, the `EXECUTE` statement must be used with the `USING` clause to correctly associate the statement parameters with the allocated SQL descriptor area or an appropriate number of host variables.

Use `INTO` for a singleton `SELECT` statement where the result set consists of only one row. For example: `SELECT COUNT(*)`

The `DESCRIBE INPUT` statement can be used to determine the `PARAMETER_MODE` of each of the parameters in the prepared statement.

If parameter markers are present, the `USING` clause must be used to specify an SQL descriptor area or a list of host variables for these parameters. There must be one variable in the host variables list or 'referenced' in the SQL descriptor area for each parameter in the prepared statement. The variables are associated with the parameter markers in a left to right manner as they appear in the prepared statement.

The data types of the variables must be compatible with their usage in the source statement.

The `descriptor-name` is identified by a host variable or a literal.

See *ALLOCATE CURSOR* on page *196* for a description of extended statements.

For a fuller discussion of the use of `EXECUTE` statements in dynamic application programs, see the *Mimer SQL Programmer's Manual, Chapter 4, Dynamic SQL*.

Restrictions

The EXECUTE statement may only be used for dynamically prepared statements.

Dynamically prepared SELECT statements and calls to result set procedures may not be executed. These should be performed using a sequence of OPEN, FETCH and CLOSE statements, i.e. using a cursor.

Note: You can use singleton SELECT statements or calls to result set procedures if the result set will contain only one row.

See the *Mimer SQL Programmer's Manual, Chapter 4, Communicating with the Application Program* for information on the format of the host variable.

Example

```
exec sql PREPARE 'stmt 1' FROM :sqlstr;
exec sql ALLOCATE DESCRIPTOR 'DescrIn';
exec sql DESCRIBE INPUT 'stmt 1' USING SQL DESCRIPTOR 'DescrIn';
...
exec sql EXECUTE 'stmt 1' USING SQL DESCRIPTOR 'DescrIn';
```

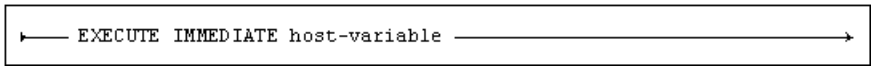
For more information, see the *Mimer SQL Programmer's Manual, Chapter 4, Executable Statements*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, "Basic dynamic SQL".

EXECUTE IMMEDIATE

Prepares and executes a dynamically submitted SQL statement.



Usage

Embedded, Module.

Description

The SQL source statement contained in the host variable is prepared and executed.

For a fuller discussion of the use of `EXECUTE` statements in dynamic application programs, see the *Mimer SQL Programmer's Manual, Chapter 4, Dynamic SQL*.

Restrictions

The `EXECUTE IMMEDIATE` statement may only be used for dynamically submitted statements with no parameter markers.

Dynamically submitted `SELECT` statements may not be executed with `EXECUTE IMMEDIATE`, these should be performed using a sequence of `OPEN`, `FETCH` and `CLOSE` statements.

Note: You can use singleton `SELECT` statements, or calls to result set procedures if the result set will contain only one row.

Example

```
strcpy(sqlstr,"DELETE FROM sometable WHERE id IS NULL");
exec sql EXECUTE IMMEDIATE :sqlstr;
```

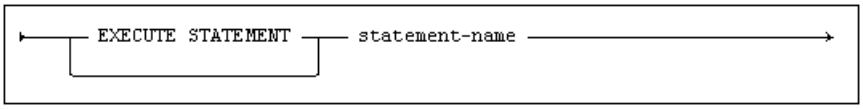
For more information, see the *Mimer SQL Programmer's Manual, Chapter 4, Executable Statements*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, "Basic dynamic SQL".

EXECUTE STATEMENT

Executes a precompiled statement.



Usage

Embedded, Interactive, Module, ODBC, JDBC

Description

The precompiled statement specified is executed. That is, the precompiled statement identified by `statement-name` is executed.

In dynamic SQL the application can inquire about the type of statement and information about input and output host variables. I.e. the precompiled statement is handled like any other statement where the actual content of the SQL statement is not known.

A precompiled statement belongs to a schema.

Restrictions

The ident executing the `EXECUTE STATEMENT` command must have execute privilege on the precompiled statement.

Examples

Interactive

```
EXECUTE STATEMENT user2.seltaba;

updtaba;
```

Embedded SQL

```
exec sql EXECUTE STATEMENT user2.seltaba;

exec sql updtaba;
```

ODBC

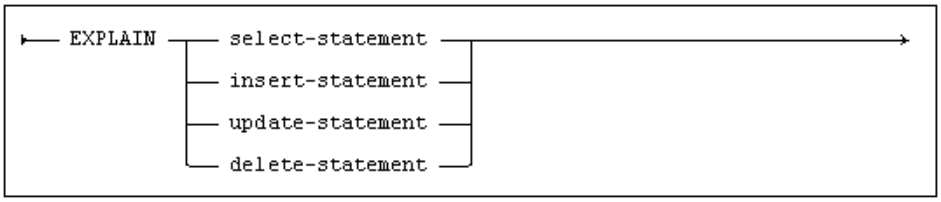
```
SQLPrepare(hStmt, "EXECUTE STATEMENT user2.seltaba", SQL_NTS);
…
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The EXECUTE STATEMENT command is a Mimer SQL extension.

EXPLAIN

Returns explain information for a statement.



Usage

Embedded, Interactive, ODBC, JDBC

Description

Explain information for a `SELECT`, `UPDATE`, `INSERT` or `DELETE` statement is returned as a result set.

The actual `SELECT`, `UPDATE`, `INSERT` or `DELETE` statement is not executed.

The `EXPLAIN` output is typically used to help in the process of constructing efficient queries.

Notes

Mimer SQL generates XML-based explain data. The `EXPLAIN` command can be used to read this data. Other ways are DbVisualizer Pro which has a graphical explain, and BSQL's explain that returns the raw XML-based output. See *Mimer SQL User's Manual, Appendix A, Mimer SQL Explain* for more information about these alternatives, and how to read and interpret the explain data.

Example

```
SQL>explain
SQL&select cou.country, cur.currency from currencies cur, countries cou
SQL&where  cou.country in ('Belgium', 'Norway')
SQL&and    cou.currency_code = cur.code;
         ID      PARENT OPERATION
OPERATIONTYPE                                SCANORDER             ACC_COST
               HITS               VISITS
TABLE
ALIAS
INDEX
INDEXONLY
================================================================================
          1           0 select
-                                                  -                     6
                   2                     6
-

-

-

-
===
          2           1 inner join
-                                                  -                     6
                   2                     6
-

-

-

-
===
          3           2 index scan, table lookup
leading keys                                       1                     4
                   2                     4
countries

cou

cnt_country_exists

FALSE
===
          4           2 table lookup
unique                                             2                     1
                   1                     1
currencies

cur

SQL_PRIMARY_KEY_0000023475

-
===

                  4 rows found
```

The above output tells it's a `SELECT` statement. `SCANORDER` 1 shows that the countries table is read first. The unique key cnt_country_exists index is used to scan the table. We have a condition on the first column in the index (`cou.country = 'Belgium'`), which is why the scan is leading keys.

The index cnt_country_exists has both the country column and the primary key code column. The `VISITS` count is 4 because two rows are read in the index, and two rows from the base table. This will result in a `HITS` count of 2 rows.

The join node contains the cost of processing the two tables.

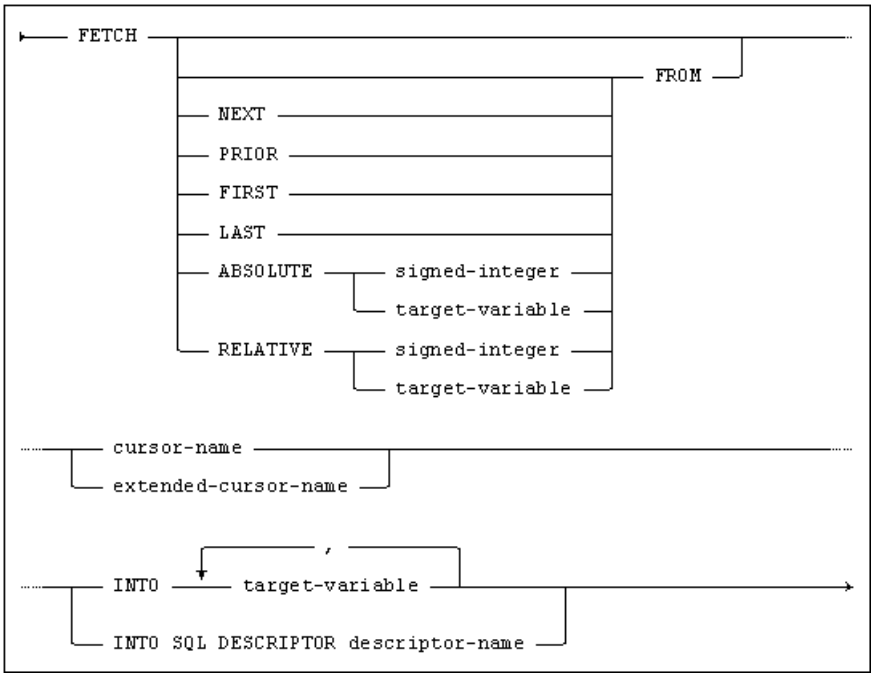
When there are no temporary tables involved the cost is equal to the total number of visits.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The EXPLAIN command is a Mimer SQL extension.

FETCH

Positions a cursor on a specified row in the result set addressed by the cursor, and assigns values from the row to host variables.



Usage

Embedded, Procedural.

Description

The named cursor is positioned on the specified row in the result set defined by the cursor declaration. This row becomes the current row for the cursor.

There are two imaginary row positions for a cursor: 'one row prior to `FIRST` and 'one row after `LAST`'. The cursor will be positioned on these rows if any of the orientation specifications cause the cursor position to move before the `FIRST` row or after the `LAST` row of the result set respectively. Once the cursor position reaches one of the imaginary rows it will not advance any further in that direction.

The `descriptor-name` is identified by a host variable or a literal.

See *ALLOCATE CURSOR* on page *196* for a description of extended cursors.

When using a scrollable cursor, the position of the cursor can be specified with an orientation specification.

The orientation can be specified in one of the following ways:

`NEXT` – Position the cursor on the row next to the current row

`PRIOR` – Position the cursor on the row prior to the current row

`FIRST` – Position the cursor on the first row in the result set

`LAST` – Position the cursor on the last row in the result set

ABSOLUTE – Position the cursor on the row with a specified absolute row number in the result set. Row zero does not exist and will return NOT FOUND. Negative numbers count from the end of the result set (i.e. -1 = LAST.)

RELATIVE – Position the cursor on the row specified with a row number relative to the current row in the result set. Zero is the current row, positive numbers count toward the end of the result set and negative numbers toward the beginning.

Values from the current row are assigned to target variables as listed in the INTO clause or specified in the named SQL descriptor area. The form FETCH… USING SQL DESCRIPTOR is used when an appropriate SQL descriptor area has been established. See the *Mimer SQL Programmer's Manual, Chapter 4, SQL Descriptor Area,* for a discussion of the use of SQL descriptor areas.

The columns retrieved from the database are defined by the SELECT clause in the cursor declaration. The value from the first column in the SELECT clause is assigned to the first variable, that from the second column to the second variable, and so on. The data type of each variable must be assignment-compatible with the value in the corresponding column.

The number of columns in a row must be the same as the number of variables specified in the INTO clause.

If there is no next row in the set of rows, the cursor is placed 'after the last row', no new values are assigned to the variables and a NOT FOUND condition code is returned, see *Appendix E Return Status and Conditions*.

Language Elements

target-variable, see *Target Variables* on page *43*.

Restrictions

SELECT access to the table or view addressed by the table reference is checked when the cursor used for the FETCH statement is opened. Access to the base table is not required for a FETCH operation on a view.

The cursor cannot be identified by specifying extended-cursor-name in a procedural usage context. The INTO SQL DESCRIPTOR clause cannot be used in a procedural usage context.

Notes

If the cursor is not declared as scrollable, the FETCH operation always positions the cursor at the next row in the result set. For such a cursor, orientation specification is not allowed (except for NEXT).

If the orientation specification is omitted for a scrollable cursor, NEXT is implicit.

If the cursor that is used by the FETCH statement is not declared with an ORDER BY clause, the sort order for the result set is undefined, even if the cursor is defined as scrollable. This means that the sort order may change if new indexes are created, if indexes are dropped, if new statistics are gathered, or if a new version of the SQL optimizer is installed. To assure a specific sort order, ORDER BY must be used.

If a data conversion error occurs in assigning a value to a variable, an error code is returned and the execution of the FETCH statement is terminated. All variables successfully assigned before the error occurred retain their assigned values.

Examples

Embedded SQL example:

```
exec sql FETCH 'Cursor1' INTO SQL DESCRIPTOR 'DescrOut';
if (strcmp(SQLSTATE,"00000") == 0) {
    exec sql GET DESCRIPTOR 'DescrOut' VALUE 1 :ival = DATA;
    …
```

PSM example:

```
…
FETCH c_2
   INTO Data;
RETURN (Data.Title, Data.Artist, Data.Format, Data.Price,
                    Data.Item_ID, Data.Artist_ID, '***');
…
```

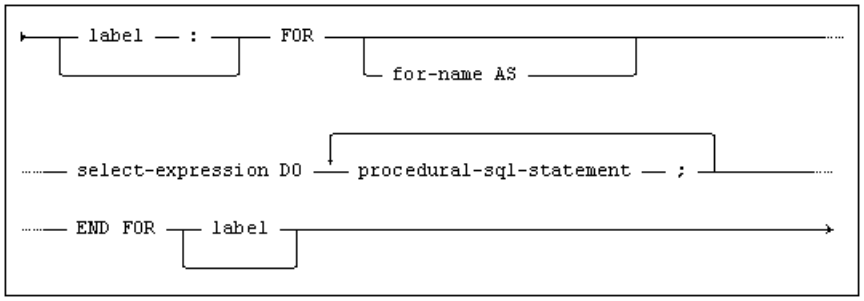
For more information on how you can use the `FETCH` statement, see the *Mimer SQL Programmer's Manual, Chapter 11, Mimer SQL Stored Procedures*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F431, "Read-only scrollable cursors".

FOR

Execute statements for each row of a result set.



Usage

Procedural.

Description

The result set defined by the query expression is iterated and for each row found the list of procedural-sql-statements in the for body is executed. All columns of the result set are available as variables within the for body. This means that all items in the select list must have a unique name. If a `for-name` is given, this can be used to qualify variable references within the for body.

The query expression can be a call to a result set procedure in which case the names in `result-set-clause` is used as variable names.

For a list of `procedural-sql-statements`, see *Procedural SQL Statements* on page *193*.

Restrictions

If labels are specified at both the beginning and the end of the `FOR` statement, they must be the same.

If a label is specified at the end it must also be specified at the beginning.

The for-name may not be the same as the name of a label for any compound statement within the scope of the for statement.

The body of a `FOR` statement is atomic which means that it cannot contain a `COMMIT`, `ROLLBACK`, or `START` statement.

Notes

A `FOR` statement may be terminated by a `LEAVE` statement using label.

Examples

```
FOR SELECT code, country, currency_code FROM countries DO
    RETURN (code, country, currency_code);
END FOR;
```

```
FOR st AS CALL mimer_store_web.view_basket(session_number) DO
    IF  st.title = 'American splendour' THEN
        ...
    ELSE
        ...
    END IF;
END FOR;
```

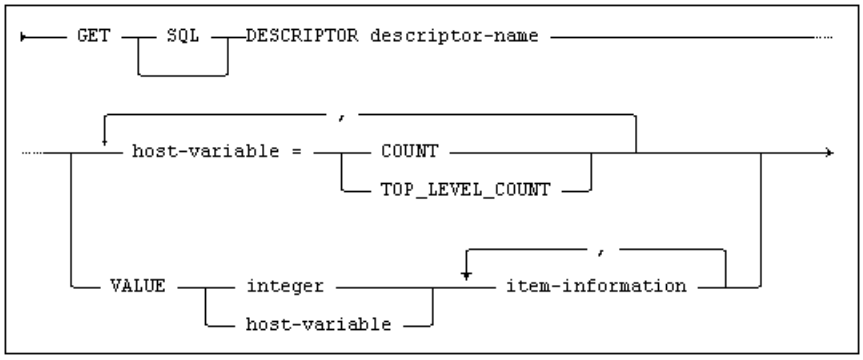
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, Iterating through a result set - FOR loop*.

Standard Compliance

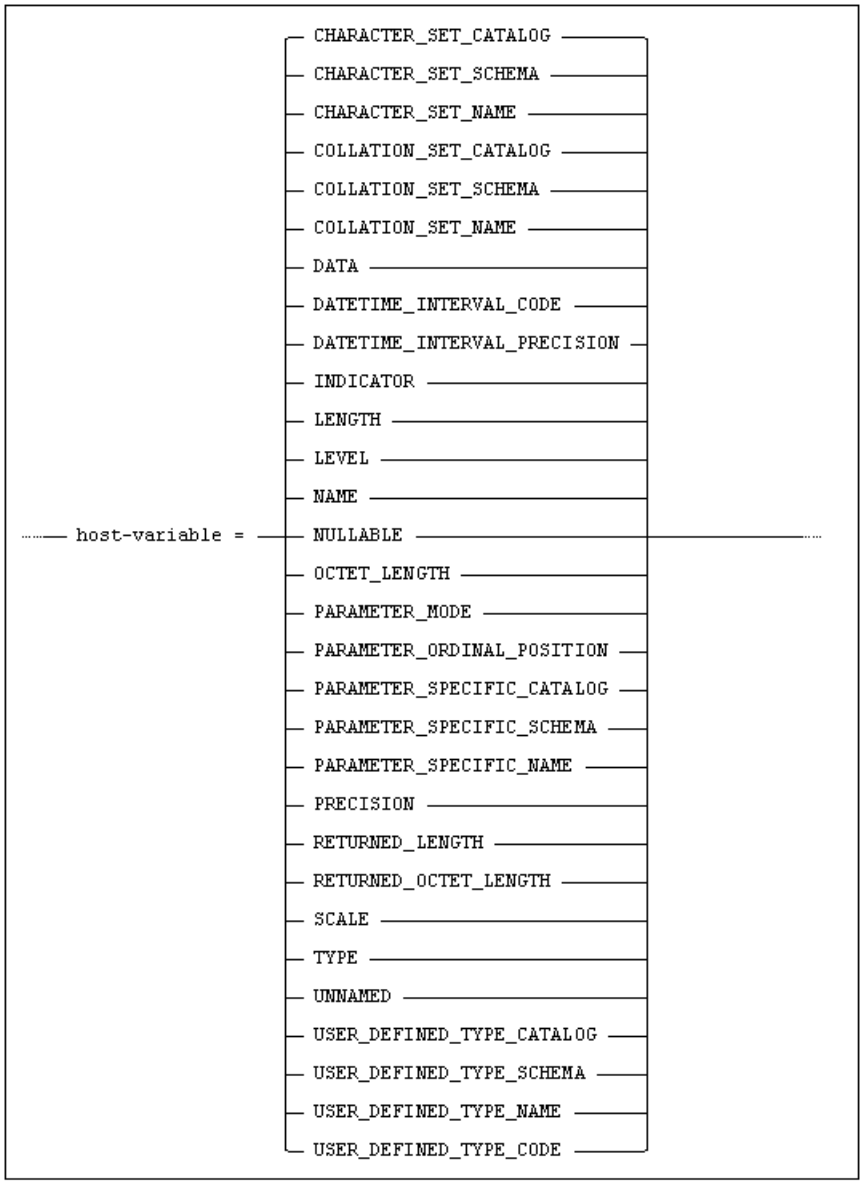
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".
	Mimer SQL extension	The support for CALL is a Mimer SQL extension.

GET DESCRIPTOR

Gets values from an SQL descriptor area.



where `item-information` is:



Usage

Embedded, Module.

Description

Values are retrieved from the specified SQL descriptor area. The GET DESCRIPTOR statement can be used in two forms:

- To determine the number of active item descriptor areas for the specified SQL descriptor, the form ...host-variable = COUNT is used. TOP_LEVEL_COUNT is used to determine the top level columns (or parameters).
- The VALUE form is used to retrieve SQL descriptor field values from the item descriptor area specified by item-number.

When using GET DESCRIPTOR with DESCRIBE OUTPUT, if COUNT > 0, the output is a result set and a cursor should be used to retrieve data.

The descriptor-name is identified by a host variable or a literal.

Item Descriptor Area

An item descriptor area contains the following fields:

Field name	Description
CHARACTER_SET_CATALOG	catalog name for the character set which the described item is using.
CHARACTER_SET_SCHEMA	A character string containing the schema name for the character set which the described item is using.
CHARACTER_SET_NAME	A character string containing the name of the character set which the described item is using.
COLLATION_CATALOG	A character string containing the catalog name for the collation which the described item is using.
COLLATION_SCHEMA	A character string containing the schema name for the collation which the described item is using.
COLLATION_NAME	A character string containing the name of the collation which the described item is using.
DATA	If the INDICATOR field does not indicate a null value, this field contains an input (OPEN or EXECUTE) or output (FETCH or EXECUTE) value with the data type specified by the TYPE field, and with the attributes specified by the applicable fields in the item descriptor area.

Field name	Description
DATETIME_INTERVAL_CODE	If the TYPE field contains 9 or 10 (i.e. for DATETIME and INTERVAL data types), this column will contain an exact numeric value with scale 0 which specifies the DATETIME or INTERVAL subtype. See below for descriptions of the codes that apply to these two data types.
DATETIME_INTERVAL_PRECISION	An exact numeric value with scale 0, which specifies the leading field precision for the INTERVAL data type (i.e. when the TYPE value is 10).
INDICATOR	An exact numeric value with scale 0, used as a null indicator for input (OPEN or EXECUTE) or output (FETCH or EXECUTE with an INTO clause) values. INDICATOR=-1 indicates a null value, and INDICATOR=0 indicates a non-null value. If INDICATOR is > 0 after a FETCH operation or an EXECUTE operation with INTO clause, it indicates that a truncation occurred and the value of INDICATOR is the required length.
LENGTH	An exact numeric value with scale 0, containing the string length of a character or binary string data type. Terminating null bytes are excluded.
LEVEL	An exact numeric value with scale 0, which identifies the item descriptor area's level. Level 0 is the top level.
NAME	A character string containing the column name, returned by DESCRIBE OUTPUT. After DESCRIBE INPUT this field contains a question mark (?).
NULLABLE	An exact numeric value with scale 0, indicating whether a resulting column can contain null or not. NULLABLE=1 indicates that null is allowed. NULLABLE=0 indicates that null is not allowed.
OCTET_LENGTH	An exact numeric value with scale 0, containing the number of octets of a character or binary string data type. Terminating null bytes are excluded.

Field name	Description
PARAMETER_MODE	An exact numeric value with scale 0, which specifies the MODE of a routine parameter. See below for a description of the code values for this field.
PARAMETER_ORDINAL_POSITION	An exact numeric value with scale 0, indicating the ordinal position of the parameter in the parameter list of the routine definition.
PARAMETER_SPECIFIC_CATALOG	A character string representing the catalog name for the invoked procedure, if the prepared statement is a call statement.
PARAMETER_SPECIFIC_SCHEMA	A character string representing the schema name for the invoked procedure, if the prepared statement is a call statement.
PARAMETER_SPECIFIC_NAME	A character string representing the name of the invoked procedure, if the prepared statement is a call statement.
PRECISION	An exact numeric value with scale 0, specifying the precision for a numeric data type value. For the data types: INTERVAL DAY TO SECOND INTERVAL HOUR TO SECOND INTERVAL MINUTE TO SECOND INTERVAL SECOND TIME and TIMESTAMP, the value in this field describes the precision of the fractional SECONDS component.
RETURNED_LENGTH	An exact numeric value with scale 0, set by FETCH or EXECUTE with an INTO clause, which returns the actual length of a VARCHAR or VARBINARY output value.
RETURNED_OCTET_LENGTH	An exact numeric value with scale 0, set by FETCH or EXECUTE with an INTO clause, which returns the actual number of octets of a VARCHAR or VARBINARY output value.
SCALE	An exact numeric value with scale 0, specifying the scale for a numeric data type value.
TYPE	An exact numeric value with scale 0, containing a coded representation of the data type. See below for a description of the codes.

Field name	Description
UNNAMED	An exact numeric value with scale 0, indicating whether NAME contains an actual column or parameter name, or not. UNNAMED=0 indicates that NAME contains an actual name. UNNAMED=1 means that NAME does not contain an actual name.

TYPE Fields in the Item Descriptor Area

The TYPE field in the item descriptor area can contain one of the following values:

Code	SQL data type
1	CHARACTER
2	NUMERIC
3	DECIMAL
4	INTEGER
5	SMALLINT
6	FLOAT
7	REAL
8	DOUBLE PRECISION
9	DATETIME
10	INTERVAL
12	VARCHAR
16	BOOLEAN
25	BIGINT
30	BLOB
40	CLOB
60	BINARY[a]
61	BINARY VARYING
-8	NCHAR
-9	NCHAR VARYING
-11	INTEGER(p)[b]
-40	NCLOB

a.Null padding is applied to the fixed size BINARY data type.

b.INTEGER(p) is a Mimer SQL specific data type used for integer data with a specified precision.

DATETIME Data Types in the Item Descriptor Area

For DATETIME data types, the DATETIME_INTERVAL_CODE field in the item descriptor area can contain one of the following values:

Code	DATETIME subtype
1	DATE
2	TIME
3	TIMESTAMP

INTERVAL Data Types in the Item Descriptor Area

For INTERVAL data types, the DATETIME_INTERVAL_CODE field in the item descriptor area can contain one of the following values:

Code	INTERVAL subtype
1	YEAR
2	MONTH
3	DAY
4	HOUR
5	MINUTE
6	SECOND
7	YEAR TO MONTH
8	DAY TO HOUR
9	DAY TO MINUTE
10	DAY TO SECOND
11	HOUR TO MINUTE
12	HOUR TO SECOND
13	MINUTE TO SECOND

Parameters in the Item Descriptor Area

For routine parameters, the PARAMETER_MODE field in the item descriptor area can contain one of the following values:

Code	PARAMETER_MODE
1	PARAMETER_MODE_IN
2	PARAMETER_MODE_INOUT
4	PARAMETER_MODE_OUT

Notes

The value of the DATA field is undefined if the INDICATOR field indicates the null value.

The data type of the host variables must be compatible with the data type of the associated field name.

Examples

```
exec sql GET DESCRIPTOR 'SQLA' :cnt = COUNT;

exec sql GET DESCRIPTOR 'SQLA' VALUE 1 :hostvar1 = DATA;
```

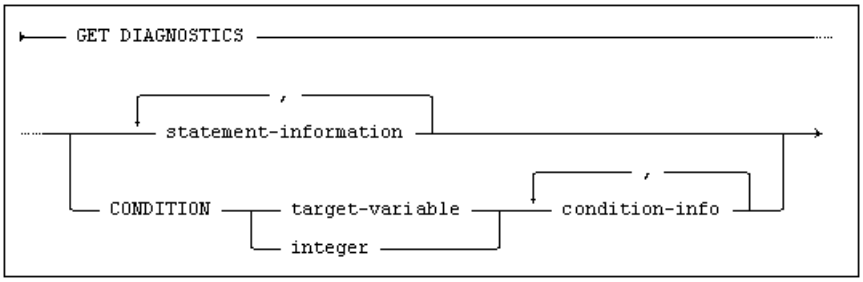
For more information, see the *Mimer SQL Programmer's Manual, Chapter 4, SQL Descriptor Area*.

Standard Compliance

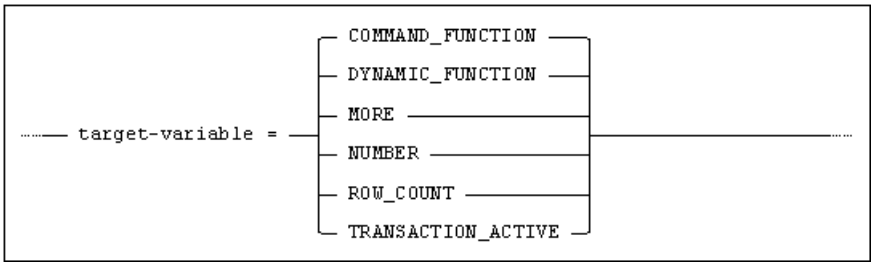
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, "Basic dynamic SQL". Feature B032, "Extended dynamic SQL" support of dynamic descriptor names.

GET DIAGNOSTICS

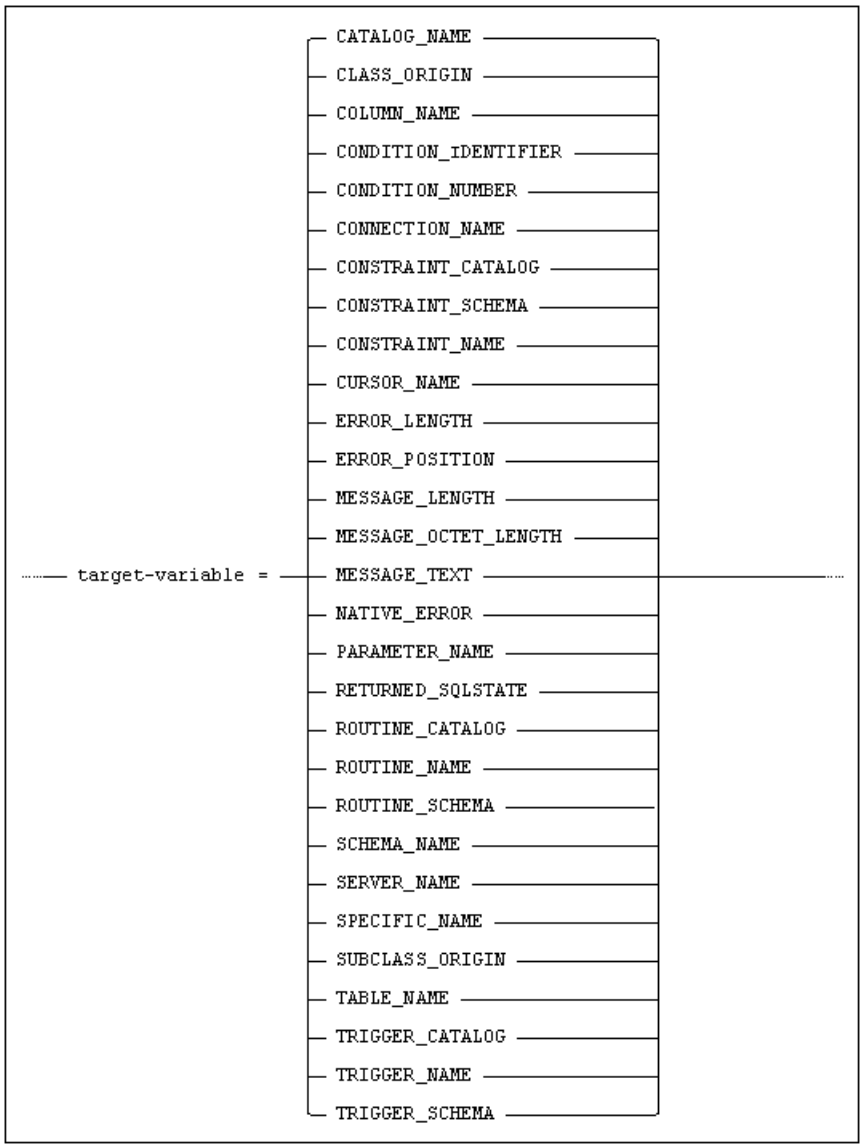
Gets statement or condition information from the diagnostics area.



where `statement-information` is:



and `condition-info` is:



Usage

Embedded, Procedural.

Description

Selected status information from the diagnostics area is retrieved. The diagnostics area holds information about the most recently executed SQL statement. There is only one diagnostics area for each application, independent of the number of connections that the application holds. Observe that the `GET DIAGNOSTICS` statement itself does not change the diagnostics area, apart from setting `SQLSTATE`.

The `GET DIAGNOSTICS` statement can be in two forms: the first form retrieves statement information about the most recent SQL statement executed. The second form of `GET DIAGNOSTICS` is the `CONDITION` form, which retrieves condition information for the most recently executed SQL statement. The ordinal number of the condition to be returned is specified immediately following the keyword `CONDITION`.

statement-information Information Items

The information items for `statement-information` are described in the following table:

Information item	Data type	Description
COMMAND_FUNCTION	NCHAR VARYING(128)	A string identifying the preceding embedded SQL statement executed.
DYNAMIC_FUNCTION	NCHAR VARYING(128)	A string identifying the preceding prepared SQL statement executed.
MORE	CHAR(1)	Indicates if there are any conditions for which no condition information has been stored. N if all detected conditions are stored in the diagnostics area, otherwise Y.
NUMBER	INTEGER	The number of condition messages stored for the most recently executed SQL statement.
ROW_COUNT	INTEGER	The number of rows inserted, updated or deleted if the last statement was INSERT, searched UPDATE or searched DELETE.
TRANSACTION_ACTIVE	INTEGER	Indicates if a transaction is active or not. 0 = transaction not active 1 = transaction is active.

condition-info Information Items

The information items for `condition-info` are described in the following table:

Information item	Data type	Description
CATALOG_NAME	NCHAR VARYING(128)	The catalog name of the schema containing the table on which the violated constraint is defined, always an empty string ("").
CLASS_ORIGIN	NCHAR VARYING(128)	The defining source of the two first characters (the class portion) of the SQLSTATE value.
COLUMN_NAME	NCHAR VARYING(128)	The name of the table column on which the violated constraint is defined. If the constraint involves more than one column or the data change operation causing the condition is not in the table on which the constraint is defined, this will be an empty string ("").

Information item	Data type	Description
CONDITION_IDENTIFIER	NCHAR VARYING(128)	The value specified for condition-name in the DECLARE CONDITION statement declaring the condition as a named condition. This will be the empty string ("") if the condition has not been declared as a named condition.
CONDITION_NUMBER	INTEGER	The ordinal number of the condition on the diagnostics condition stack.
CONNECTION_NAME	NCHAR VARYING(128)	The connection name specified in a CONNECT, DISCONNECT or SET CONNECTION statement. The name of the current connection for all other statements.
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The catalog name of the schema containing the violated constraint, always an empty string ("").
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the violated constraint.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the violated constraint.
CURSOR_NAME	NCHAR VARYING(128)	The name of the cursor which is in an invalid state, when the condition: 24000 - 'Invalid Cursor State' is raised.
ERROR_LENGTH	INTEGER	The length in characters of the relevant part the SQL statement, starting at ERROR_POSITION.
ERROR_POSITION	INTEGER	The position in the SQL statement where the specified condition occurred. Value < 1 means unknown position.
MESSAGE_LENGTH	INTEGER	The length of the message text for the specified condition.
MESSAGE_OCTET_LENGTH	INTEGER	Currently the same as MESSAGE_LENGTH.
MESSAGE_TEXT	NCHAR VARYING(254)	The descriptive message text for the specified condition.

Information item	Data type	Description
NATIVE_ERROR	INTEGER	The internal Mimer SQL return code relating to the condition. See the *Mimer SQL Programmer's Manual, Appendix B, Return Codes*.
PARAMETER_NAME	NCHAR VARYING(128)	The name of the routine parameter causing the condition.
RETURNED_SQLSTATE	CHAR(5)	Value of SQLSTATE for the specified condition.
ROUTINE_CATALOG	NCHAR VARYING(128)	The catalog name of the schema containing the function or procedure in which the condition was raised, always an empty string ("").
ROUTINE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the function or procedure in which the condition was raised.
ROUTINE_NAME	NCHAR VARYING(128)	The name of the function or procedure in which the condition was raised.
SCHEMA_NAME	NCHAR VARYING(128)	The name of the schema containing the table on which the violated constraint is defined. If the data change operation causing the condition is not in the table on which the constraint is defined, this will be an empty string ("").
SERVER_NAME	NCHAR VARYING(128)	The database name specified in a CONNECT, DISCONNECT or SET CONNECTION statement. The current database name for all other statements.
SPECIFIC_NAME	NCHAR VARYING(128)	Specific name of the procedure or function in which the condition was raised.
SUBCLASS_ORIGIN	NCHAR VARYING(128)	The defining source of the three last characters (the subclass portion) of the SQLSTATE value.

Information item	Data type	Description
TABLE_NAME	NCHAR VARYING(128)	The name of the table on which the violated constraint is defined. If the data change operation causing the condition is not in the table on which the constraint is defined, this will be an empty string ("").
TRIGGER_CATALOG	NCHAR VARYING(128)	The catalog name of the schema containing the table supporting the trigger in which the condition was raised, always an empty string ("").
TRIGGER_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table supporting the trigger in which the condition was raised.
TRIGGER_NAME	NCHAR VARYING(128)	The name of the trigger in which the condition was raised.

Values for COMMAND_FUNCTION and DYNAMIC_FUNCTION

The COMMAND_FUNCTION and DYNAMIC_FUNCTION information items can contain any of the following values:

```
ALLOCATE CURSOR                DROP STATEMENT
ALLOCATE DESCRIPTOR            DROP SYNONYM
ALTER DATABANK                 DROP TABLE
ALTER DATABANK RESTORE         DROP TRIGGER
ALTER IDENT                    DROP TYPE
ALTER SHADOW                   DROP VIEW
ALTER STATEMENT                DYNAMIC CLOSE
ALTER TABLE                    DYNAMIC DELETE CURSOR
ALTER TYPE                     DYNAMIC FETCH
ASSIGNMENT                     DYNAMIC OPEN
CALL                           DYNAMIC UPDATE CURSOR
CLOSE CURSOR                   ENTER
COMMENT                        EXECUTE
COMMIT WORK                    EXECUTE IMMEDIATE
CONNECT                        FETCH
CREATE BACKUP                  GET DESCRIPTOR
CREATE COLLATION               GET DIAGNOSTICS
```

```
CREATE DATABANK
CREATE DOMAIN
CREATE FUNCTION
CREATE IDENT
CREATE INDEX
CREATE METHOD
CREATE MODULE
CREATE PROCEDURE
CREATE SCHEMA
CREATE SEQUENCE
CREATE SHADOW
CREATE STATEMENT
CREATE SYNONYM
CREATE TABLE
CREATE TRIGGER
CREATE TYPE
CREATE VIEW
DEALLOCATE DESCRIPTOR
DEALLOCATE PREPARE
DELETE CURSOR
DELETE WHERE
DESCRIBE
DISCONNECT
DROP COLLATION
DROP DATABANK
DROP DOMAIN
DROP FUNCTION
DROP IDENT
DROP INDEX
DROP METHOD
DROP MODULE
DROP PROCEDURE
DROP SCHEMA
GRANT
GRANT OBJECT PRIVILEGE
GRANT SYSTEM PRIVILEGE
INSERT
LEAVE
LEAVE RETAIN
OPEN
PREPARE
REVOKE
REVOKE OBJECT PRIVILEGE
REVOKE SYSTEM PRIVILEGE
ROLLBACK WORK
SELECT
SET CONNECTION
SET DATABANK
SET DATABASE
SET DESCRIPTOR
SET SESSION DIAGNOSTIC SIZE
SET SESSION ISOLATION LEVEL
SET SESSION READ ONLY
SET SESSION READ WRITE
SET SESSION START EXPLICIT
SET SESSION START IMPLICIT
SET SHADOW
SET TRANSACTION DIAGNOSTIC SIZE
SET TRANSACTION ISOLATION LEVEL
SET TRANSACTION READ ONLY
SET TRANSACTION READ WRITE
SET TRANSACTION START EXPLICIT
SET TRANSACTION START IMPLICIT
START TRANSACTION
UPDATE CURSOR
UPDATE STATISTICS
```

```
DROP SEQUENCE                   UPDATE WHERE

DROP SHADOW
```

Language Elements

`target-variable`, see *Target Variables* on page *43*.

Notes

The condition requested by the `GET DIAGNOSTICS CONDITION` form must be one of the conditions that exist in the diagnostics area, i.e. the condition number must be in the range from 1 up to the value of `NUMBER`.

Example

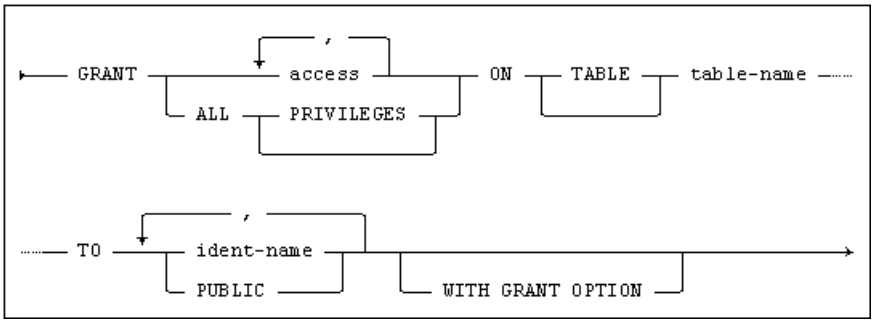
```
…
exec sql GET DIAGNOSTICS :cnt = NUMBER;
for (int i = 1; i <= cnt; i++) {
    exec sql GET DIAGNOSTICS CONDITION :i
        :sqlstatestr = RETURNED_SQLSTATE,
        :errmsgstr = MESSAGE_TEXT,
        :errmsglen = MESSAGE_LENGTH;
    …
}
…
```

Standard Compliance

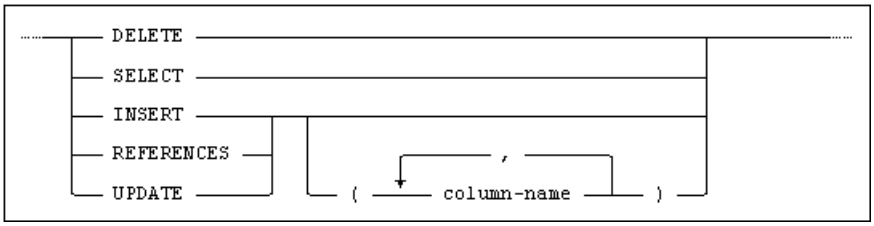
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F121, “Basic diagnostics management”.
	Mimer SQL extension	The support for NATIVE_ERROR, ERROR_LENGTH and ERROR_POSITION is a Mimer SQL extension.

GRANT ACCESS PRIVILEGE

Grants one or more access privileges on a table or view.



where `access` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The specified access privileges are granted to the `ident`(s) listed. When `WITH GRANT OPTION` is specified, the ident may in turn grant the specified access privileges to another ident. If the privileges are granted to a `GROUP` ident, all members of the group receive the privileges.

Access Privileges

The `access` privileges are as follows:

- `DELETE`

  allows the ident to delete rows from the table or view identified by `table-name`. If a selective delete is specified, involving a `WHERE` clause, appropriate privileges must also to held to permit execution of the search-condition, otherwise the delete operation will fail.

- `INSERT`

  allows the ident to insert new rows into the table or view identified by `table-name`. If a `column-name` list is supplied, the access is restricted to the specified columns, otherwise the access applies to the entire table (including any new columns subsequently added).

- REFERENCES

  allows the ident to use the table identified by table-name in the FOREIGN KEY clause of a CREATE TABLE statement. REFERENCES access may only be granted on a base table, not on a view. If a column-name list is supplied, the access is restricted to the specified columns, otherwise the access applies to the entire table (including any new columns subsequently added).

- SELECT

  Allows the ident to select rows from the table or view identified by table-name.

- UPDATE

  Allows the ident to update the table or view identified by table-name. If a column-name list is supplied, the access is restricted to the specified columns, otherwise the access applies to the entire table (including any new columns subsequently added). If a selective update is specified, involving a WHERE clause, appropriate privileges must also to held to permit execution of the search-condition, otherwise the update operation will fail.

Access privileges may be granted in any combination. Specification of the keyword ALL (followed by the optional keyword PRIVILEGES) instead of an explicit list of access privileges results in all applicable privileges being granted to the specified ident(s) (i.e. all privileges which the grantor is authorized to grant).

Restrictions

The grantor must have grant option on the access privilege.

Notes

If the grantor loses WITH GRANT OPTION, any access privileges he has granted using it are automatically revoked.

An ident may not grant access privileges to himself.

Example

The following example is taken from the *Mimer SQL User's Manual, Chapter 8, Granting Access Privileges*.

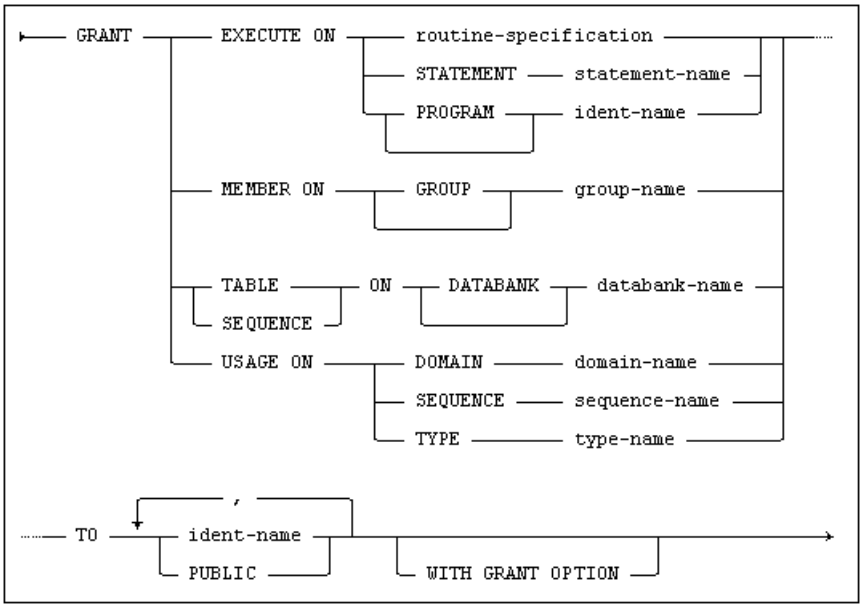
```
GRANT SELECT, UPDATE(exchange_rate) ON currencies TO mimer_admin_group;
```

Standard Compliance

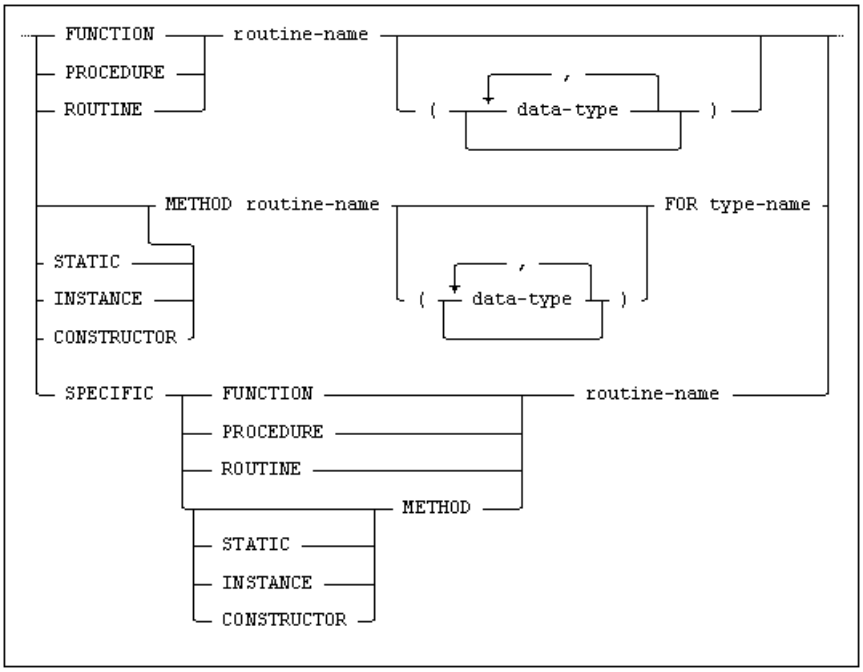
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F731, "INSERT column privileges" support for granting insert on individual columns.
	Mimer SQL extension	The keyword PRIVILEGES is optional and not mandatory in Mimer SQL.

GRANT OBJECT PRIVILEGE

Grants object privileges to one or more idents.



where `routine-specification` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The specified object privilege is granted to the ident(s) listed. When `WITH GRANT OPTION` is specified, the ident may in turn grant the specified privilege to another ident. If the privilege is granted to a `GROUP` ident, all members of the group receive the specified privilege.

Object Privileges

The object privileges are as follows:

- `EXECUTE`

  Allows the ident to invoke the specified function, procedure or precompiled statement, or to enter the specified `PROGRAM` ident.

- `MEMBER`

  Specifies that the ident is a member of the stated group. All privileges granted to the group are held by all members of the group.

- `SEQUENCE`

  Allows the ident to create new sequences in the specified databank.

- `TABLE`

  Allows the ident to create new tables in the specified databank.

- `USAGE`

  Grant usage on a domain or user defined type allows the ident(s) to use the domain or the user defined type where a data type would normally be specified. This includes, amongst others, table definitions, routine definitions and cast expression.

  When a grant usage on type statement is executed, this will also incur that the ident(s) will receive execute on all functions, on which the grantor has execute privilege with grant option, that were created implicitly when the user defined type was created.

  Grant usage on a sequence allows the ident(s) to use the specified sequence in the next value for and current value for expressions.

Restrictions

The grantor must have grant option on the privilege.

Notes

If the grantor loses his grant option, any privileges he has granted using the option are automatically revoked.

An ident may not grant privileges to himself.

Example

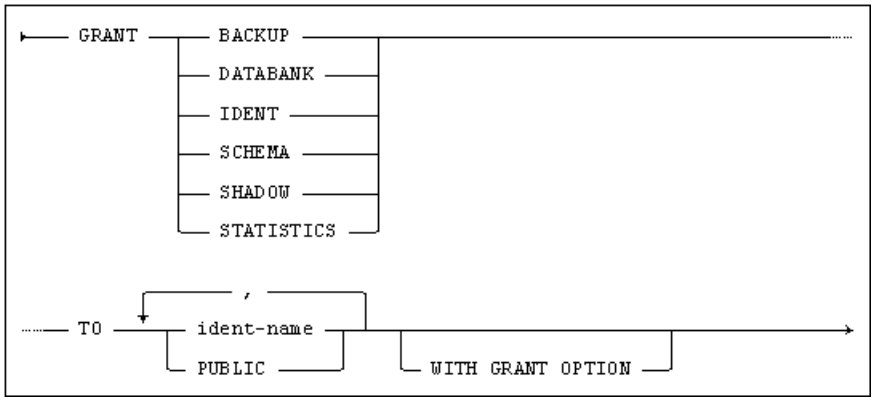
```
GRANT EXECUTE ON FUNCTION capitalize TO mimer_admin_group;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F251, “Domain support” support for grant usage on domain. Feature F690, “Collation support” support for grant usage on collation.
	Mimer SQL extension	Support for use of MEMBER, EXECUTE (on STATEMENT and PROGRAM), SEQUENCE and TABLE are Mimer SQL extensions.

GRANT SYSTEM PRIVILEGE

Grants system privileges to one or more idents.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The specified system privilege is granted to the ident(s) listed. When `WITH GRANT OPTION` is specified, the ident may in turn grant the specified privilege to another ident. If the privilege is granted to a `GROUP` ident, all members of the group receive the specified privilege.

System Privileges

The system privileges are as follows:

- `BACKUP`

  Enables the ident to perform databank backup and restore operations.

- `DATABANK`

  Enables the ident to create new databanks. The databank file is created by the Mimer SQL system. The privilege authorizes the ident to create files, using the file access used by the database server process, in the operating system.

- `IDENT`

  Enables the ident to create new Mimer SQL idents and schemas.

- `SCHEMA`

  Enables the ident to create new schemas.

- `SHADOW`

  Enables the ident to create and perform operations on databank shadows.

- `STATISTICS`

  Enables the ident to execute the `UPDATE STATISTICS` and `DELETE STATISTICS` statements.

Restrictions

The grantor must have grant option on the privilege.

Notes

An ident may not grant privileges to himself.

Example

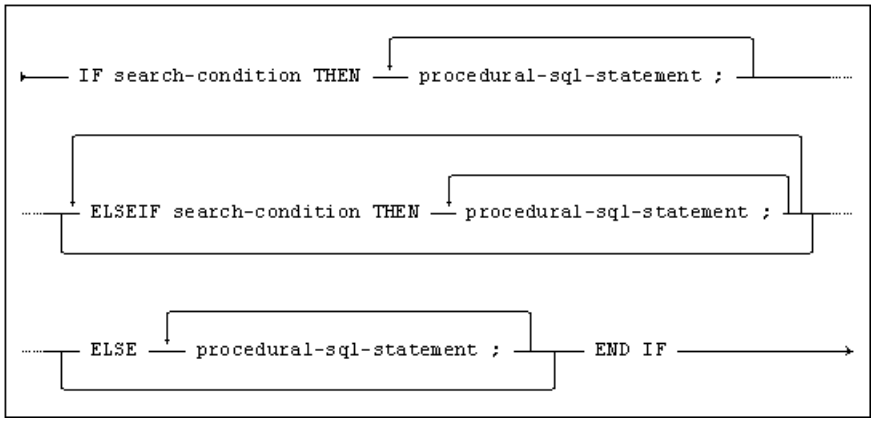
```
GRANT IDENT TO mimer_adm WITH GRANT OPTION;
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The grant system privileges is a Mimer SQL extension.

IF

Provides conditional execution based on the truth value of a conditional expression.



Usage

Procedural.

Description

The `IF` statement allows a sequence of `procedural-sql-statements` to be conditionally executed depending on the truth value of a `search-condition`.

For a list of `procedural-sql-statements`, see *Procedural SQL Statements* on page *193*.

All of the predicates supported by Mimer SQL may be used in the `search-condition`, see *Predicates* on page *153*.

In a basic `IF` statement, the sequence of `procedural-sql-statements` in the `THEN` clause will be executed if `search-condition` evaluates to true, otherwise the sequence of `procedural-sql-statements` in the `ELSE` clause will be executed.

One or more `IF` statements can be nested by using the `ELSEIF` clause in place of an `ELSE` clause containing another `IF` statement.

Language Elements

`search-condition`, see *Search Conditions* on page *165*.

Notes

If the `search-condition` equals null or directly includes the null value, it evaluates to unknown and its treated as false. If it is required that the conditional expression test for the null value, then the correct behavior is achieved by using the `IS NULL` predicate, see *The NULL Predicate* on page *159*.

Example

```
if  X > 50 then
    SET X = 50;
    SET Y = 1;
else
    SET Y = 0;
end if;

declare bookExists boolean;

set bookExists = exists (select * from books where ... );

if  bookExists then
     ...
end if;

declare bookTitle row(ISBN varchar(20), Title varchar(50));
...

if  booktitle = ('0-201-43328-1','JDBC API Tutorial and Reference') then
     ...
end if;
```

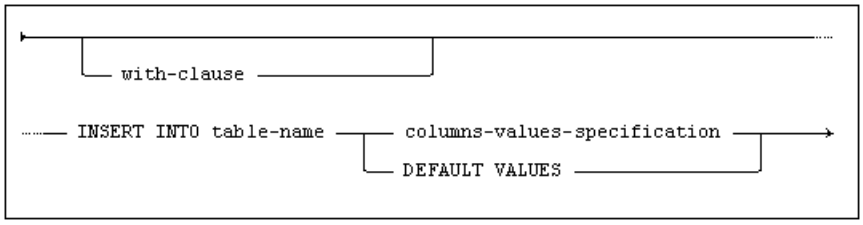
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, Conditional Execution Using IF*.

Standard Compliance

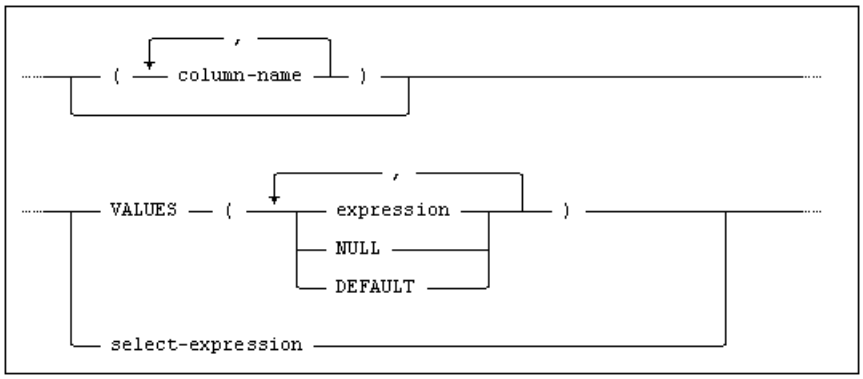
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".

INSERT

Inserts one or more rows into a table or view.



where `columns-values-specification` is:



Usage

Embedded, Interactive, Module, ODBC, Procedural, JDBC.

Description

One or more new rows are inserted into the table or view specified in `table-name`.

If a list of column names is given in `columns-values-specification`, only the specified columns are assigned values in accordance with the INSERT statement.

The columns not listed are assigned their default value or the null value in accordance with the column definition, see *CREATE TABLE* on page *285*. If `table-name` specifies a view, any columns in the base table which are excluded from the view are also assigned their default value or the null value in the same way.

If the column name list is omitted, all columns in the table or view are implicitly specified in the order in which they are defined in the table or view. This practice is, however, not recommended when `INSERT` statements are embedded in application programs, since the semantics of the statement will change if the table or view definition is changed.

Specification of a `DEFAULT VALUES` clause inserts a single row into the table with the column default value specified for each column in the table.

Values are assigned in order from the items in the `VALUES` clause or the select-specification to the columns that have been explicitly or implicitly specified. The number of values specified must be the same as the number of columns and the data type of each value must be assignment-compatible with the column into which it is to be inserted.

Specification of a VALUES clause inserts a single row into the table or view. The keyword NULL or DEFAULT can be specified in the VALUES clause to insert the null value or the column default value, respectively, into the corresponding column.

Specification of a select-specification instead of a VALUES clause inserts the set of rows resulting from the select-specification into the target table or view. If the set of rows resulting from the select-specification is empty, a NOT FOUND condition code is returned, see *Appendix E Return Status and Conditions*.

Language Elements

expression, see *Chapter 9, Expressions and Predicates*.

select-specification, see *Chapter 11, The SELECT Expression*.

with-clause, see *The WITH Clause* on page *179*.

Restrictions

INSERT access is required on the table or view specified in the INTO clause.

If a select-specification is specified, SELECT access is required on the table(s) from which the selection is performed.

In a procedural usage context, the INSERT statement is only permitted if the procedure access-clause is MODIFIES SQL DATA, see *CREATE PROCEDURE* on page *271*.

Notes

Expressions used in the VALUES clause cannot refer to column names or set functions.

If the row or rows inserted do not conform to constraints imposed on the table, no rows are inserted. Constraints are as follows:

- Values in the primary key and unique keys of the base table may not be duplicated. This also applies to unique secondary indexes.
- FOREIGN KEY constraints must be observed.
- CHECK constraints in table, column and domain definitions must be observed for insertions.
- For insertion into views defined WITH CHECK OPTION, inserted values must conform to the view definition.

Example

```
INSERT INTO countries (country_code, country, currency_code) VALUES
    ('CX', 'Christmas Island', 'AUD');
```

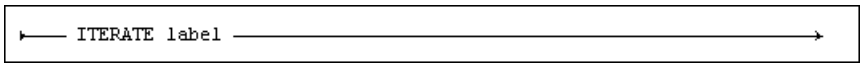
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F222, “INSERT statement: Default values clause.” Feature F781, “Self referencing operations” the table in the insert clause may occur in the query specification.

ITERATE

Continues execution at the beginning of a labeled `procedural-sql-statement`.



Usage

Procedural.

Description

The `ITERATE` statement can be used to skip the remaining statements within a labeled `procedural-sql-statement`. The execution will continue at the beginning of the labeled `procedural-sql-statement`. The statement must be a `FOR`, `LOOP`, `REPEAT` or `WHILE` statement.

For a list of `procedural-sql-statements`, see *Procedural SQL Statements* on page *193*.

The label is the beginning label of a `procedural-sql-statements` within the scope containing the `ITERATE` statement.

Restrictions

A `procedural-sql-statement` must have a beginning label if `ITERATE` is to be used.

Notes

If the `ITERATE` statement is contained in any compound statements which are enclosed in the `procedural-sql-statement` defined by the label the following actions will take place:

- Every open cursor declared in the compound statements is closed.
- All variables, cursors and handlers declared in the compound statements are destroyed.
- All condition names declared in the compound statements cease to be defined.

Example

```
L1:
LOOP
  ...

    ITERATE L1;
  ...
END LOOP L1;
```

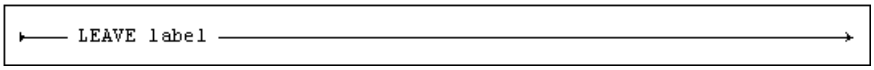
For more information, see *Mimer SQL Programmer's Manual, Chapter 11, Iteration*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, “Computational completeness”.

LEAVE

Leaves a labeled `procedural-sql-statement`.



Usage

Procedural.

Description

The `LEAVE` statement can be used to terminate the execution of a `procedural-sql-statement`. For a list of procedural SQL statements, see *Procedural SQL Statements* on page *193*.

The `label` is the beginning label of a `procedural-sql-statement` within the scope containing the `LEAVE` statement.

Restrictions

A `procedural-sql-statement` must have a beginning label if `LEAVE` is to be used to terminate its execution.

Notes

All intervening compound statements between the statement identified by the label and the leave statement will be left. The following actions occur before execution of the compound statements is terminated, after the `LEAVE` statement is executed:

- Every open cursor declared in the compound statements is closed.
- All variables, cursors and handlers declared in the compound statements are destroyed.
- All condition names declared in the compound statements cease to be defined.

If the `LEAVE` statement is executed within a compound statement forming the body of a procedure, execution of the procedure will be terminated.

Example

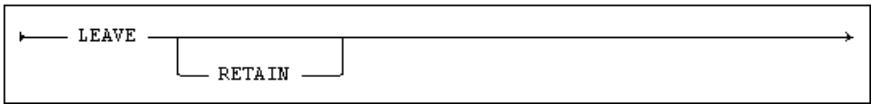
```
CREATE PROCEDURE procedure_name(INOUT Y INTEGER)
CONTAINS SQL
S0:
BEGIN
  ...
  S1:
  BEGIN
    IF Y < 0 THEN
      SET Y = 0;
      LEAVE S0;
    END IF;
    ...
  END S1;
  ...
END S0;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, “Computational completeness”.

LEAVE (PROGRAM ident)

Leaves a `PROGRAM` ident and restores the state of the previous ident.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The current `PROGRAM` ident is left and the saved environment of the previous ident is restored.

If `RETAIN` is specified, resources allocated to the ident being left are kept, i.e. cursor declarations. The cursors are however inactivated, and may not be used in any statement until the same ident is re-entered.

Restrictions

The `LEAVE` statement may not be issued within a transaction.

Example

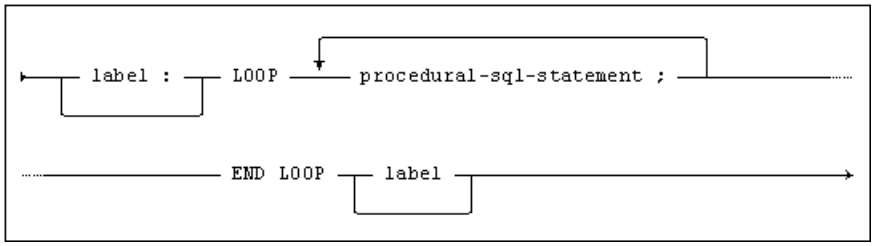
```
ENTER 'program_name' USING 'secret';
LEAVE RETAIN;
```

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The LEAVE (program) statement is a Mimer SQL extension.

LOOP

Allows one or more `procedural-sql-statements` to be iteratively executed.



Usage

Procedural.

Description

The `LOOP` statement can be used to iteratively execute a sequence of one or more procedural SQL statement.

For information on `procedural-sql-statements`, see *Procedural SQL Statements* on page *193*.

Restrictions

If `label` appears at the beginning and at the end of the `LOOP` statement, the same value must be specified in both places.

Specifying `label` is optional, however, if `label` appears at the end of the `LOOP` statement, it must also appear at the beginning.

A `label` is required at the beginning if the `LEAVE` statement is to be used to terminate the LOOP statement.

Notes

The `LOOP` statement itself does not include a mechanism for terminating the iteration.

The `LOOP` statement can be terminated by executing the `LEAVE` statement. The `LOOP` statement will also terminate if an exception condition is raised, in accordance with the normal exception handling process.

The `LOOP` statement does not establish any form of local scope, as the compound statement does, the `label` is only specified to allow the `LEAVE` statement to be used to terminate the iteration.

Example

```
L1:
LOOP
  ...

    LEAVE L1;
  ...
END LOOP L1;
```

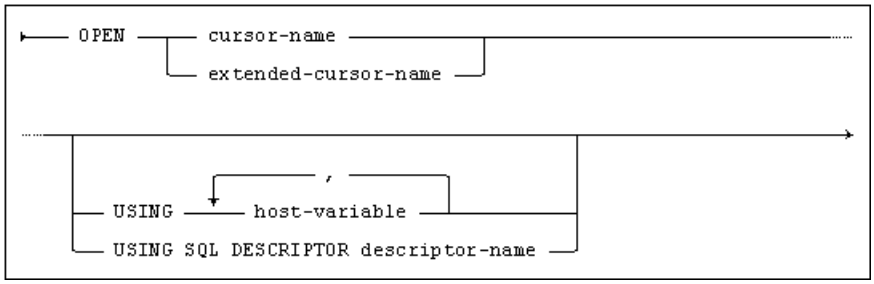
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, Iteration*

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".

OPEN

Opens a cursor.



Usage

Embedded, Procedural.

Description

The cursor is opened and references a set of rows in accordance with the definition of the cursor. The result set is defined when the `OPEN` statement is executed, any inserts, updates or deletes occurring after the open will not be reflected in the result set. The cursor is placed 'before' the first row in the addressed set.

The `descriptor-name` is identified by a host variable or a literal.

See *ALLOCATE CURSOR* on page *196* for a description of extended cursors.

If the cursor is declared for a dynamically prepared `SELECT` statement, the source form of which contains parameter markers, the `OPEN` statement must include a `USING` clause.

The first variable in the variable list or referenced in the descriptor area takes the place of the first parameter marker, the second variable takes the place of the second marker, and so on.

The number of variables provided in the `USING` clause must be equal to the number of parameter markers in the source statement, and the data types of the variables must be assignment-compatible with their usage in the source statement. See the *Mimer SQL Programmer's Manual, Chapter 4, Dynamic SQL,* for a more detailed description of the use of dynamically prepared statements.

Open cursors can be closed using one of the statements `CLOSE`, `COMMIT` or `ROLLBACK`, except for cursors declared `WITH HOLD` which remain open after `COMMIT`.

An open cursor must be closed before it can be opened again, unless it is declared as `REOPENABLE`.

A cursor declared as `REOPENABLE` can have several open instances of a cursor. The state of the current instance is stored on a cursor stack when a new instance is opened. The state of the preceding cursor instance is restored when a cursor is closed, see *CLOSE* on page *237*. In this context, the state of a cursor instance includes both the set of rows addressed by the cursor and the position of the cursor within the set.

Restrictions

SELECT access is required to the table(s) or view(s) addressed by the cursor.

EXECUTE access is required to the result set procedure addressed by the cursor.

In a procedural usage context, extended-cursor-name cannot be used to identify the cursor and neither of the USING options may be used.

Notes

A cursor must be declared with a DECLARE CURSOR statement or allocated with an ALLOCATE CURSOR statement before it may be opened.

All access rights that the current ident holds to the table(s) or view(s) addressed by the cursor are checked when the cursor is opened.

If SELECT access is lacking, the OPEN statement will fail.

If UPDATE or DELETE access is lacking, the cursor may be opened but any corresponding UPDATE CURRENT or DELETE CURRENT statements will fail.

Only cursors declared for dynamically prepared statements may be opened with a USING clause.

Example

```
DECLARE c_1 CURSOR FOR SELECT product, producer, format,
                               extract_date(release_date), price, item_id
                         FROM product_details
…
…
OPEN c_1;
LOOP
    FETCH c_1 INTO c1_row;
…
END LOOP;
CLOSE c_1;
```

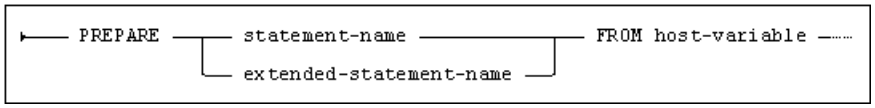
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, Using Cursors*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

PREPARE

Prepares an SQL statement contained in a host string variable for execution.



Usage

Embedded, Module.

Description

The SQL statement contained in the host variable is prepared for execution. This function is equivalent to pre-processing and compiling an embedded SQL statement, but is performed at run-time. The (internal) output form of the statement is identified by the `statement-name` parameter.

See *ALLOCATE CURSOR* on page *196* for a description of extended statements.

See also, *DESCRIBE* on page *323*.

The following SQL statements can be used with the `PREPARE` statement:

```
ALTER              EXECUTE STATEMENT  SET DATABASE
CALL               GRANT              SET SESSION
COMMENT            INSERT             SET SHADOW
COMMIT             LEAVE              SET TRANSACTION
CREATE             REVOKE             START
DELETE             ROLLBACK           UPDATE
DELETE CURRENT     SELECT             UPDATE CURRENT
DROP               SET
ENTER              SET DATABANK
```

Notes

The source form of the SQL statement to be prepared is not preceded by the identifier `EXEC SQL` or terminated by the language-specific delimiter.

The source statement may contain question marks as parameter markers to represent variables to be used when the prepared statement is executed. See the *Mimer SQL Programmer's Manual, Chapter 4, Dynamic SQL,* for more details.

Example

```
...
EXEC SQL BEGIN DECLARE SECTION;
    CHARACTER SQL_TXT(255);
...
EXEC SQL END DECLARE SECTION;
...
SQL_TXT := "DELETE FROM products WHERE product_search LIKE ?";
EXEC SQL PREPARE statement1 FROM :SQL_TXT;
...
```

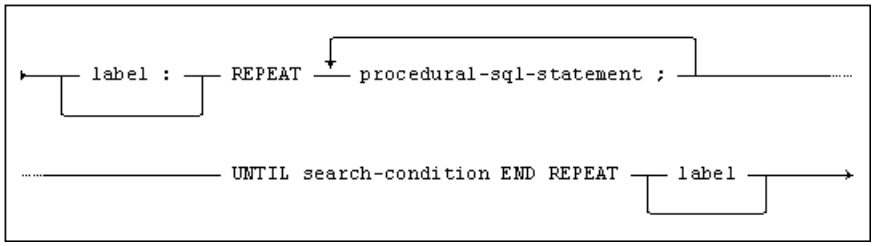
For more information, see the *Mimer SQL Programmer's Manual, Chapter 4, Preparing Statements*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, "Basic dynamic SQL".

REPEAT

Allows one or more `procedural-sql-statements` to be iteratively executed.



Usage

Procedural.

Description

The `REPEAT` statement can be used to iteratively execute a sequence of one or more `procedural-sql-statements`.

The iteration continues until the `search-condition` evaluates to true.

For information on `procedural-sql-statements`, see *Procedural SQL Statements* on page *193*.

Restrictions

If `label` appears at the beginning and at the end of the `REPEAT` statement, the same value must be specified in both places.

Specifying `label` is optional, however, if `label` appears at the end of the `REPEAT` statement, it must also appear at the beginning.

A `label` is required at the beginning if the `LEAVE` statement is to be used to terminate the `REPEAT` statement.

Notes

The `REPEAT` statement may be terminated by executing the `LEAVE` statement using `label`. It will also terminate if an exception condition is raised, in accordance with the normal exception handling process.

Example

```
SET I = 0;
L1:
REPEAT
  ...
  SET I = I + 1;
UNTIL I > 10
END REPEAT L1;
```

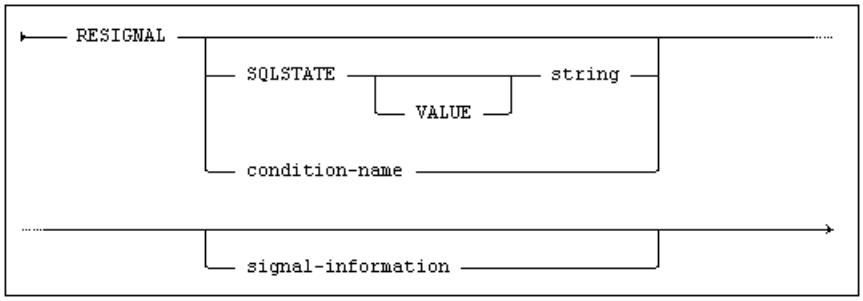
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, Iteration Using REPEAT*.

Standard Compliance

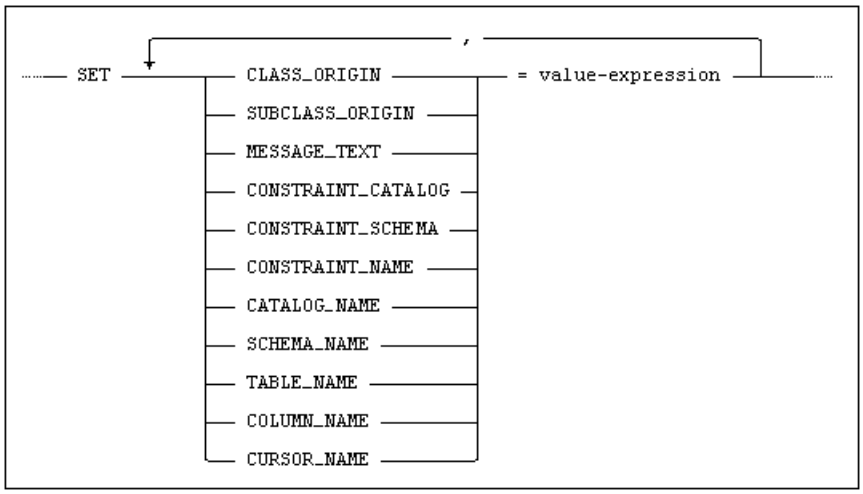
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, “Computational completeness”.

RESIGNAL

Raises the current, or the specified, exception condition.



where *signal-information* is:



Usage

Procedural.

Description

The `RESIGNAL` statement has the effect of raising the current exception condition, if specified without an argument, or an alternative exception condition specified by either an `SQLSTATE` value or a condition name.

If a condition identifier is used in the statement the associated `SQLSTATE` value is raised. If the condition identifier is declared without an `SQLSTATE` value, the `SQLSTATE 45000` is raised. If there is an appropriate exception handler for this `SQLSTATE`, this handler will be invoked otherwise the `SQLSTATE` is propagated to the calling environment.

It is possible to provide diagnostics information with the resignal statement. This diagnostics information can be retrieved where the exception is handled. This can be used to customize error messages for an application. The signal-information fields are described in the GET DIAGNOSTICS section *condition-info Information Items* on page *353*.

Note that an `SQLSTATE` value corresponding to a warning condition that is not catched by an exception handler will not be propagated to the calling environment.

Restrictions

The `RESIGNAL` statement may only be used within an exception handler, see *DECLARE HANDLER* on page *312*, to force re-propagation of an exception condition to the scope or calling environment enclosing the scope supporting the exception handler.

Notes

See *DECLARE CONDITION* on page *307* for a description of how to declare a condition name.

Example

```
RESIGNAL SQLSTATE 'UE456';
```

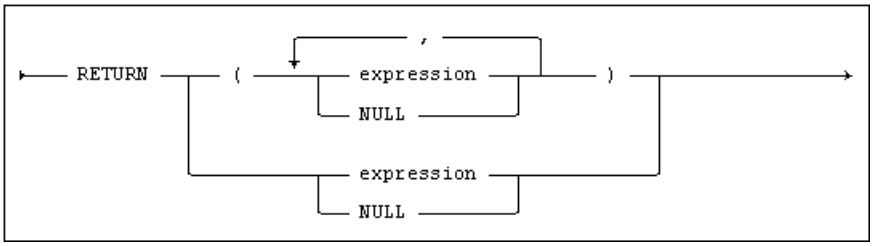
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, RESIGNAL Statements*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".

RETURN

Returns the specified value(s) from a result set procedure or a function.



Usage

Procedural.

Description

The RETURN statement is used in a function to return the single value of the function.

The SQL statements in the body of the function are executed until a RETURN statement is executed. If the end of the function is encountered (because no RETURN statement has been executed) an exception is raised.

The RETURN statement is used in a result set procedure to return the value(s) of a row of the result set to the calling cursor when FETCH is executed for it.

When a FETCH is executed for a cursor calling a result set procedure, the SQL statements in the body of the result set procedure are executed until a RETURN statement is executed, then execution within the result set procedure is suspended until the next FETCH.

Note: An array FETCH will cause more than one RETURN statement to be executed, so there is not necessarily a 1:1 correspondence between the number of FETCH statements executed and the number of RETURN statements executed.

If, following a FETCH, the end of the result set procedure is encountered instead of a RETURN statement, the NOT FOUND exception is raised to indicate the end of the result set.

Restrictions

If the RETURN statement is used in a procedure, it must be a result set procedure, see the *Mimer SQL Programmer's Manual, Chapter 11, Result Set Procedures*.

Notes

If only one value expression is being returned, the parentheses are optional.

Example

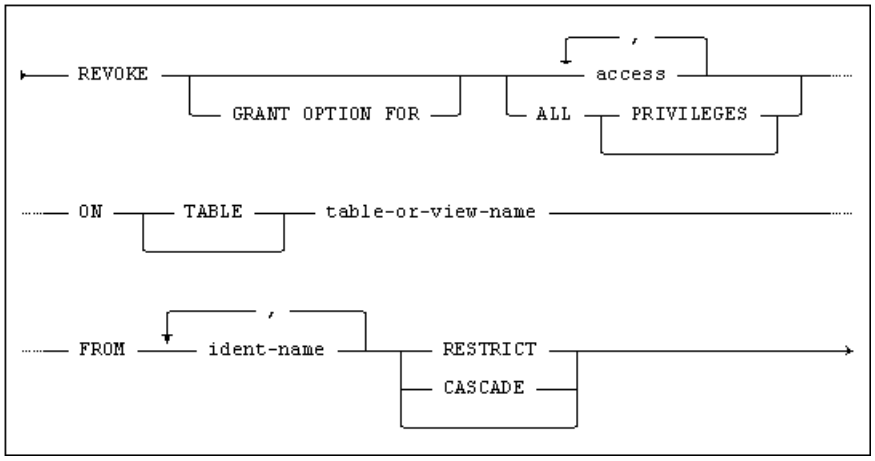
```
CREATE FUNCTION SQUARE_INTEGER(ROOT INTEGER)
RETURNS INTEGER
CONTAINS SQL
BEGIN
    RETURN ROOT*ROOT;
END
```

Standard Compliance

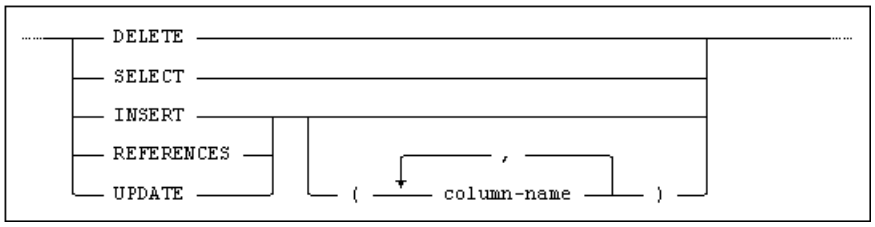
Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

REVOKE ACCESS PRIVILEGE

Revokes access privileges on a table or view, from one or more idents.



where `access` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The specified access privileges are revoked from the ident(s) listed. If the privileges are revoked from a `GROUP` ident, all members of the group lose the privileges.

The access privileges are described under `GRANT ACCESS PRIVILEGE`.

The access privileges may be revoked in any combination. Specification of the keyword `ALL` (followed by the optional keyword `PRIVILEGES`) instead of an explicit list of privileges results in all access privileges on the table or view being revoked from the specified ident(s).

The `GRANT OPTION FOR` clause specifies that only the `WITH GRANT OPTION` is to be revoked from the specified instance(s) of the privilege(s).

The keywords `CASCADE` and `RESTRICT` specify whether the `REVOKE` statement will allow the recursive effects that cause views to be dropped or `FOREIGN KEY` constraints to be removed, as a result of the `REVOKE` statement. Refer to the Notes section below for details of the recursive effects. If `CASCADE` is specified, such recursive effects will be allowed. If `RESTRICT` is specified, the `REVOKE` statement will return an error if it would cause such recursive effects and then no access privileges will be revoked.

If neither `CASCADE` nor `RESTRICT` is specified, then `RESTRICT` is implicit.

Restrictions

Privileges may only be revoked explicitly by the grantor.

Notes

If an access privilege has been granted to the same ident more than once (by different grantors), the REVOKE statement will only revoke (or will revoke the WITH GRANT OPTION from) the single instance of the privilege that was granted by the current ident.

The access rights attached to the privilege (or the WITH GRANT OPTION) will only be lost when the last instance of the privilege has been revoked.

Revoking access privileges has recursive effects.

When SELECT access on a table or view is revoked, views based on that table or view and created under the authorization of that access, are recursively dropped.

When UPDATE, INSERT, DELETE or REFERENCES access on a table or view is revoked, the same privilege on views based on that table or view and created under the authorization of the access are recursively revoked.

When REFERENCES access on an entire table or on one or more explicitly specified columns of the table is revoked, any FOREIGN KEY constraints in tables created by that ident under the authorization of that privilege are removed.

When INSERT, REFERENCES or UPDATE access is revoked from one or more explicitly specified columns of a table or view, the same privilege on columns of views based on that table or view and created under the authorization of the access are recursively revoked.

Revoking INSERT, REFERENCES or UPDATE access from one or more explicitly specified columns of a table or view will not affect access held on other column(s) of that table or view. If the original access was granted on the entire table or view, the access will stay in effect at the table level and will, therefore, apply to any new columns added to the table.

When the last instance of the required access held by the creator of a routine or trigger on a table is revoked, any routines or triggers created by that ident which contain references to the table will be dropped.

When the last instance of a privilege WITH GRANT OPTION is revoked, all instances of the privilege granted by the ident under that authorization are recursively revoked.

An ident may not revoke access privileges from itself.

Example

```
REVOKE INSERT ON countries FROM joe RESTRICT;
```

For more information, see the *Mimer SQL User's Manual, Chapter 8, Revoking Access Privileges*,

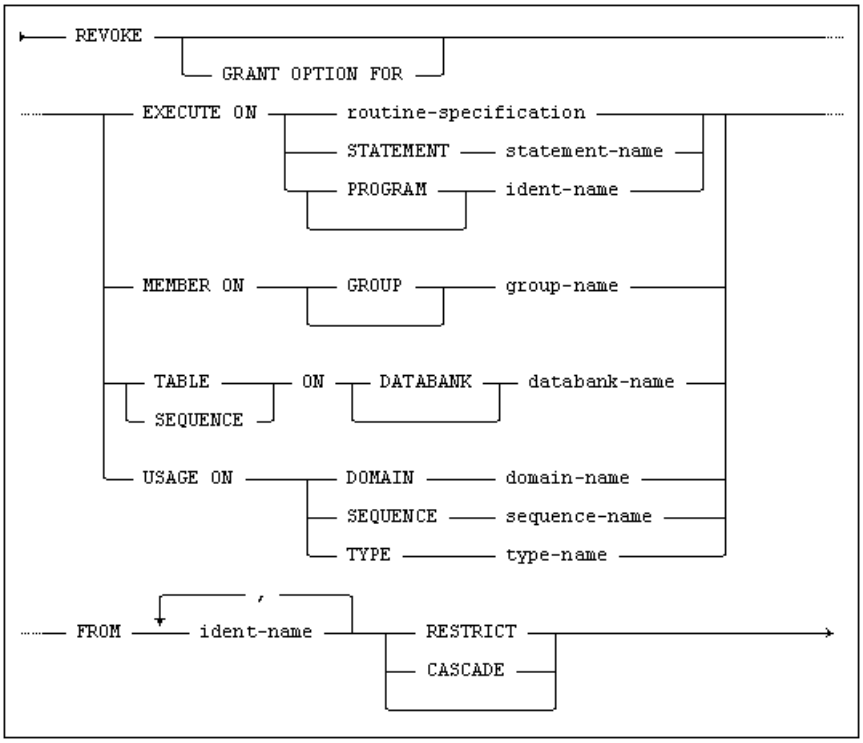
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

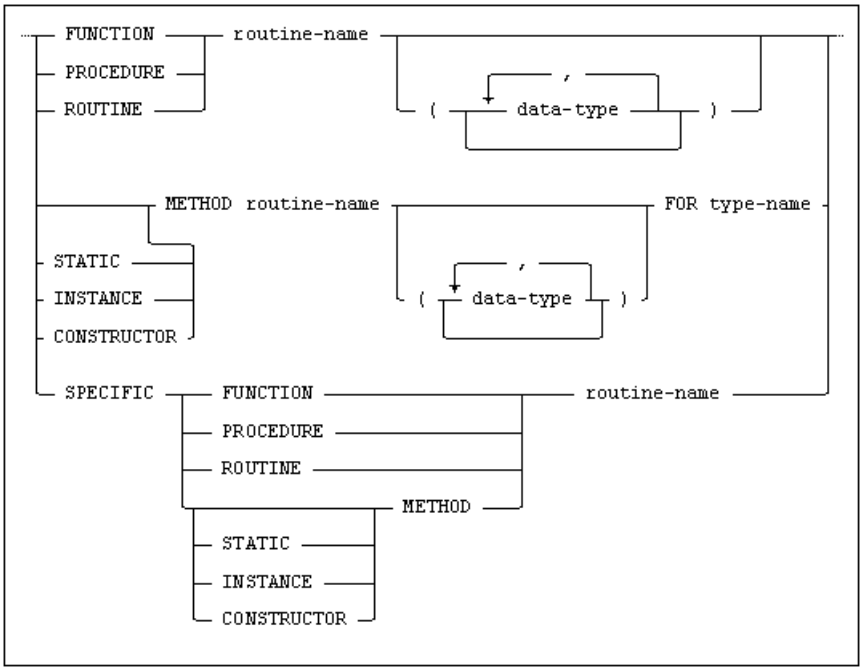
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F034, “Extended REVOKE statement” support for REVOKE CASCADE and REVOKE GRANT OPTION FOR.
	Mimer SQL extension	The keywords RESTRICT/CASCADE are optional in Mimer SQL.

REVOKE OBJECT PRIVILEGE

Revokes object privileges from one or more idents.



where `routine-specification` is:



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The specified object privilege is revoked from the ident(s) listed. If the privilege is revoked from a GROUP ident, all members of the group lose the privilege.

The object privileges are described under GRANT OBJECT PRIVILEGE.

The GRANT OPTION FOR clause specifies that only the WITH GRANT OPTION is to be revoked from the specified instance(s) of the privilege(s).

The keywords CASCADE and RESTRICT specifies whether the REVOKE statement will allow recursive effects that causes views to be dropped or FOREIGN KEY constraints to be removed, because access privileges are revoked as a result of a REVOKE MEMBER statement. See the Notes section for *REVOKE ACCESS PRIVILEGE* on page *388* for a description of when views are dropped and FOREIGN KEY constraints are removed due to recursive effects.

If CASCADE is specified, such recursive effects will be allowed.

If RESTRICT is specified, the REVOKE statement will return an error if it would cause such recursive effects, and no access privileges will be revoked.

If neither CASCADE nor RESTRICT is specified, then RESTRICT is implicit.

Restrictions

Privileges may only be explicitly revoked by the grantor.

Notes

If an object privilege has been granted to the same ident more than once (by different grantors), the REVOKE statement will only revoke (or will revoke the WITH GRANT OPTION from) the single instance of the privilege that was granted by the current ident.

The object rights attached to the privilege (or the WITH GRANT OPTION) will only be lost when the last instance of the privilege has been revoked.

Revoking object privileges has recursive effects. Privileged actions are performed under the authorization of the most recently granted instance of the access.

When the last instance of a privilege WITH GRANT OPTION is revoked, all instances of the privilege granted by the ident under that authorization are recursively revoked.

If MEMBER privilege on a group is revoked from an ident, all privileges granted through the group are revoked from the ident.

An ident may not revoke privileges from himself.

Revoking TABLE privilege does not drop the tables created when the privilege was held.

Revoking SEQUENCE privilege does not drop the sequences created when the privilege was held.

Revoking USAGE privilege on a domain, user defined type or sequence preserves the uses of the domain, user defined type or sequence which were set up when the privilege was held, however, new instances of usage of the domain, user defined type or sequence are prevented.

Revoking `USAGE` privilege on a user defined type will also revoke `EXECUTE` privilege on any function for which `EXECUTE` was granted implicitly in conjunction with executing a `GRANT USAGE ON TYPE` statement. (See *GRANT OBJECT PRIVILEGE* on page *361*.)

Revoking `EXECUTE` privilege immediately prevents the ident from invoking the routine or entering the `PROGRAM` ident.

Example

```
REVOKE EXECUTE ON PROCEDURE coming_soon FROM joe, jane;
```

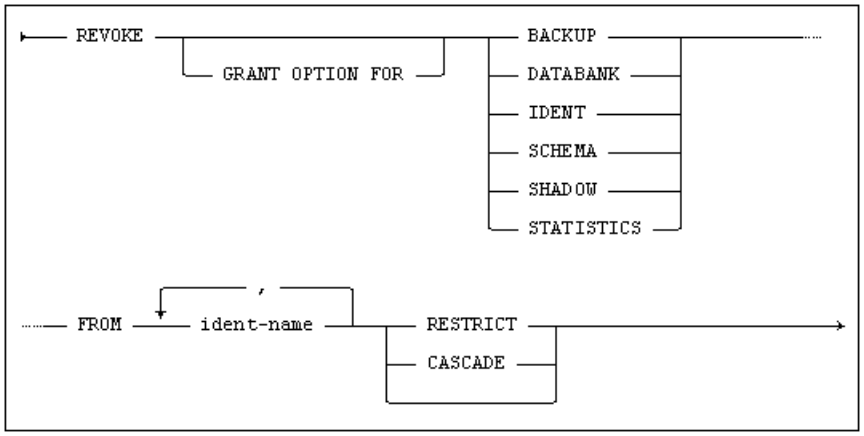
For more information, see the *Mimer SQL User's Manual, Chapter 8, Revoking Object Privileges*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F034, "Extended REVOKE statement" support for REVOKE CASCADE and REVOKE GRANT OPTION FOR. Feature F251, "Domain support" support for revoke usage on domain. Feature F690, "Collation support" support for revoke usage on collation.
	Mimer SQL extension	Revoke MEMBER, revoke SEQUENCE, revoke TABLE and revoke EXECUTE (on statement and program) are Mimer SQL extensions

REVOKE SYSTEM PRIVILEGE

Revokes system privileges from one or more idents.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The specified system privilege is revoked from the ident(s) listed. If the privilege is revoked from a `GROUP` ident, all members of the group lose the privilege.

The system privileges are described under `GRANT SYSTEM PRIVILEGE`.

The `GRANT OPTION FOR` clause specifies that only the `WITH GRANT OPTION` is to be revoked from the specified instance(s) of the privilege.

The keywords `CASCADE` and `RESTRICT` specifies whether the `REVOKE` statement will allow recursive effects if the privilege was given with grant option. If `CASCADE` is specified recursive effects will be allowed, else an error will be returned and the privilege is not revoked.

If neither `CASCADE` nor `RESTRICT` is specified, then `RESTRICT` is implicit.

Restrictions

Privileges can only be revoked explicitly by the grantor.

Notes

If a system privilege has been granted to the same ident more than once (by different grantors), the `REVOKE` statement will only revoke (or will revoke the `WITH GRANT OPTION` from) the single instance of the privilege that was granted by the current ident.

The system rights attached to the privilege (or the `WITH GRANT OPTION`) will only be revoked when the last instance of the privilege has been revoked.

Revoking system privileges has recursive effects on instances of the privilege being granted to other idents by virtue of the `WITH GRANT` option.

When the last instance of a privilege `WITH GRANT OPTION` is revoked, all instances of the privilege granted by the ident under that authorization are recursively revoked.

Databanks, idents, shadows or schemas created while the privilege was held are not dropped when the privilege is revoked.

An ident may not revoke privileges from itself.

Example

```
REVOKE DATABANK FROM joe, jane RESTRICT;
```

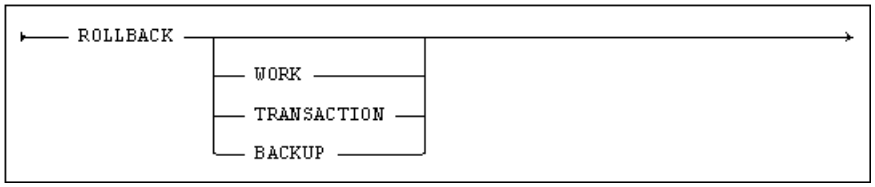
For more information, see the *Mimer SQL User's Manual, Chapter 8, Revoking System Privileges*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	Revoke system privileges is a Mimer SQL extension.

ROLLBACK

Aborts the current transaction.



Usage

Embedded, Interactive, Module, Procedural.

Description

The current transaction is aborted. No database alterations requested in the transaction build-up are executed.

All cursors opened by the current connection are closed, even those declared as `WITH HOLD`.

If there is no currently active transaction, any cursors opened by the current ident are closed, but the `ROLLBACK` statement is otherwise ignored. No error code is returned in this case.

When a `BACKUP` transaction is rolled back, all files created with `CREATE BACKUP` are deleted.

Restrictions

The `ROLLBACK` statement cannot be used in a result set procedure.

The `ROLLBACK` statement cannot be used within an atomic compound SQL statement, see *COMPOUND STATEMENT* on page *243*.

The `ROLLBACK BACKUP` statement must be used to abort a `BACKUP` transaction.

The `ROLLBACK BACKUP` statement is not supported in procedural usage contexts.

Notes

See the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security,* for a more detailed discussion of transaction handling.

Example

```
exec sql INSERT INTO sometable VALUES (:hv1, :hv2...);
if  SQLSTATE = "00000" then
    exec sql COMMIT;
else
    exec sql ROLLBACK;
end if;
```

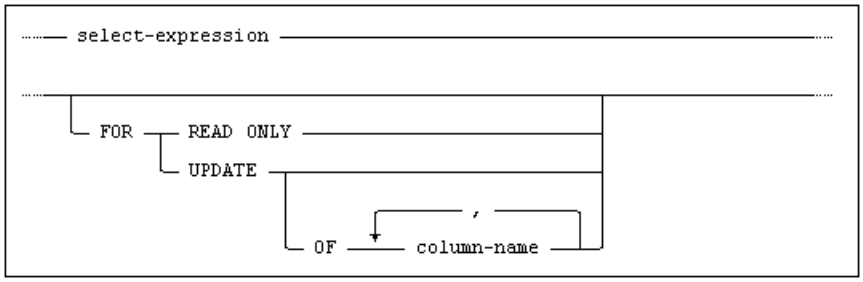
For more information, see the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Control Statements*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
	Mimer SQL extension	The use of the keywords BACKUP and TRANSACTION is a Mimer SQL extension.

SELECT

Retrieves data from the tables in the database.



Usage

Embedded, Interactive, Module, ODBC, Procedural, JDBC.

In ESQL, the `SELECT` statement may only be used to declare a cursor or as input to a `PREPARE` statement.

In a procedural usage context, the `SELECT` statement may only be used to declare a cursor.

In interactive SQL, the `SELECT` statement is used for interactive data retrieval. See the *Mimer SQL User's Manual, Chapter 3, Retrieving Data* for more details.

Description

SELECT Statements

Simple `SELECT` statements are built from a `select-expression`, see *Chapter 11, The SELECT Expression*, optionally followed by a `FOR UPDATE OF` clause.

`SELECT` statements are used in embedded SQL (including procedural usage contexts) to define cursors and as the input to dynamic `PREPARE` statements.

The embedded `SELECT` statement is syntactically equivalent to the interactive data retrieval `SELECT` statement. In embedded contexts however, the statement cannot be used to retrieve data directly but must be implemented through a cursor.

The FOR UPDATE Clause

If the `SELECT` statement defines a cursor intended for `UPDATE CURRENT` statements, the `for-update` clause must be specified. If the `FOR UPDATE OF` version is used, it must include all the columns to be updated.

Each column specified in the `for-update-of` must belong to the table or view named in the `from` clause of the `SELECT` statement, although the columns in FOR UPDATE OF do not need to be specified in the `select` clause. No column may be named more than once in the `for-update-of` clause.

Column names in the `for-update-of` clause may not be qualified by the name of the table or view. They are implicitly qualified by the table reference in the `from` clause of the select specification.

FOR UPDATE OF may not be specified if the statement defines a read-only result set, see *Updatable Result Sets* on page *399*.

Updatable Result Sets

A result set is only updatable if all of the following conditions are true (otherwise the result set is read-only):

- the keyword DISTINCT is not specified
- there are no set-functions in the SELECT list (AVG, COUNT, MAX, MIN, SUM)
- the FROM clause specifies exactly one table reference and that table reference refers either to a base table or an updatable view
- the result set is not the product of an explicit INNER or OUTER JOIN
- the GROUP BY clause is not included
- the HAVING clause is not included
- the keyword EXCEPT is not included
- the keyword INTERSECT is not included
- the keyword UNION is not included
- the ORDER BY clause is not included
- it is not the result of a call to a result set procedure
- The FOR UPDATE clause has been specified.

A cursor which addresses a read-only result table may not be used for DELETE CURRENT or UPDATE CURRENT statements.

The FOR READ ONLY Clause

The FOR READ ONLY clause is optional since SELECT statements by default are read-only.

Examples

```
SELECT format, category_id
FROM formats
ORDER BY LOWER(format), category_id;

SELECT format AS format_name, category
FROM formats
ORDER BY CASE category WHEN 'ROCK' THEN 1
                      WHEN 'JAZZ' THEN 2
                      ELSE 3
         END
OFFSET 10 ROWS FETCH FIRST 5 ROWS ONLY;
```

List all artists and use the FETCH FIRST construction to pick one arbitrary album for each artist.

```
SELECT a.artist,
       (SELECT p.product
        FROM mimer_store.products AS p
          JOIN mimer_store.items AS i ON p.product_id = i.product_id
          JOIN mimer_store_music.titles AS t ON i.item_id = t.item_id
        WHERE t.artist_id = a.artist_id
        FETCH FIRST 1 ROW ONLY) AS work_sample
FROM mimer_store_music.artists AS a;
```

Find the 10 most common starts of artist names, leading 'The' excluded:

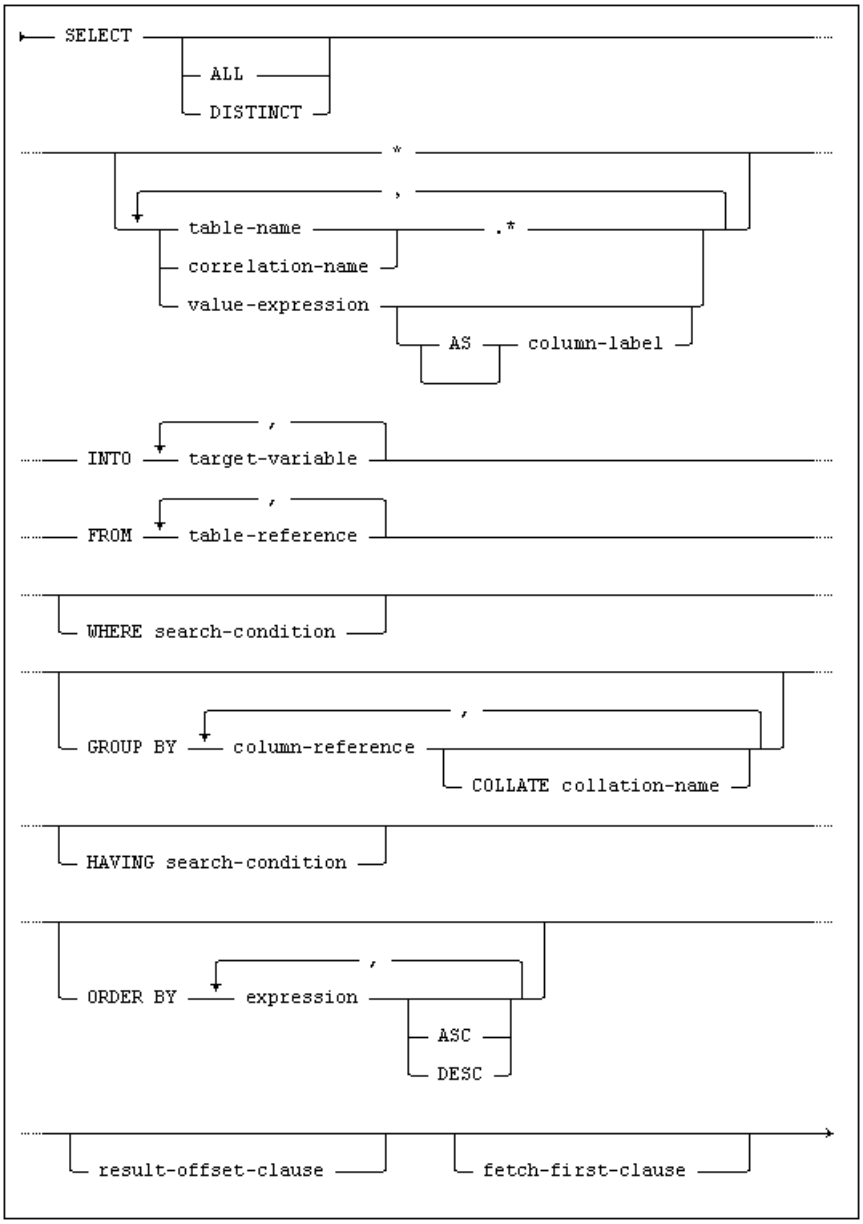
```
SELECT strt, count(*) AS cnt
FROM
(
    SELECT CASE WHEN artist NOT LIKE 'The %' THEN CAST(artist AS nchar(3))
                ELSE SUBSTRING(artist FROM 5 FOR 3)
           END AS strt
    FROM mimer_store_music.artists
) AS a
GROUP BY str
ORDER BY cnt DESC
FETCH FIRST 10 ROWS ONLY;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F302, "INTERSECT table operator". Feature F304, "EXCEPT ALL table operator". Feature F851, "<order by clause> in subqueries". Feature F855, "Nested <order by clause> in <query expression>". Feature F856," Nested <fetch first clause> in <query expression>". Feature F857, "Top-level <fetch first clause> in <query expression>" Feature F858, "<fetch first clause> in subqueries". Feature F860, "dynamic <fetch first row count> in <fetch first clause>". Feature F861, "Top-level <result offset clause> in <query expression>". Feature F862, "<result offset clause> in subqueries". Feature F865, "dynamic <offset row count> in <result offset clause>". Feature T551, "Optional keywords for default syntax" support for the keyword DISTINCT.
	Mimer SQL Extension	Support for host variable in <fetch first clause> and <result offset clause> is a Mimer SQL extension.

SELECT INTO

Selects a single-row result table and assigns the values directly to host variables. Also known as a singleton `SELECT`.



Usage

Embedded, Procedural.

Description

Values defined by the `SELECT`, `FROM` and `WHERE` clauses are assigned to target variables as specified in the `INTO` clause. The value of the first element in the `SELECT` clause is assigned to the first variable, the value of the second element to the second variable, and so on. The data types of the variables must be assignment-compatible with those of the corresponding values.

The number of elements in the select-list must be the same as the number of elements in the target-variable list.

The result table defined by the SELECT INTO statement may not contain more than one row.

If a table reference or correlation name is used together with an asterisk in the SELECT clause, all columns are selected from the referred table. Columns listed explicitly in the SELECT clause need not be prefixed with the table or view name unless the same column name is used in more than one source table or view.

The whole list of values in the SELECT clause may be replaced by a single asterisk, in which case all columns from the table(s) or view(s) named in the FROM clause are selected, in the order in which they are defined in the source table(s) or view(s).

Note: Use of SELECT * is discouraged in are embedded in application programs (except in EXISTS predicates) since the asterisk is expanded to a column list when the statement is compiled, and any subsequent alterations in the table or view definitions may cause the program to function incorrectly.

When set functions are used in the list of values in the SELECT clause, their evaluation is influenced by the keywords ALL and DISTINCT. If ALL is specified, all rows in the result table are used in calculating the result of the set function. If DISTINCT is specified, duplicate rows are eliminated from the result table before the set function is evaluated. If no keyword is specified, ALL is assumed.

Language Elements

expression, see *Expressions* on page *141*.

search-condition, see *Search Conditions* on page *165*.

target-variable, see *Target Variables* on page *43*.

order-by-clause, see *The ORDER BY Clause* on page *186*.

result-offset-clause, see *The RESULT OFFSET Clause* on page *186*.

fetch-first-clause, see *The FETCH FIRST Clause* on page *187*.

Restrictions

SELECT access is required on all tables and views specified in the statement.

In a procedural usage context, the SELECT INTO statement is only permitted if the routine access-clause is READS SQL DATA or MODIFIES SQL DATA, see *CREATE FUNCTION* on page *258* and *CREATE PROCEDURE* on page *271*.

Notes

Correlation names used in the SELECT or WHERE clause must be defined in the FROM clause of the same SELECT INTO statement. The same correlation name may not be defined more than once in one FROM clause.

A SELECT INTO statement may include a GROUP BY or HAVING clause. However, care must be exercised to ensure that the HAVING clause selects one and only one group, and that the selected group either contains only one member or is reduced to a single row by a set function.

Examples

Use the SUM aggregate function to make sure exactly one row is returned:

```
SELECT SUM(quantity * "VALUE") INTO :hv
FROM mimer_store.items AS msi
JOIN mimer_store.order_items AS msoi ON msi.item_id = msoi.item_id
WHERE order_id = :inval;
```

Use FETCH FIRST 1 ROW ONLY to make sure exactly one row is returned:

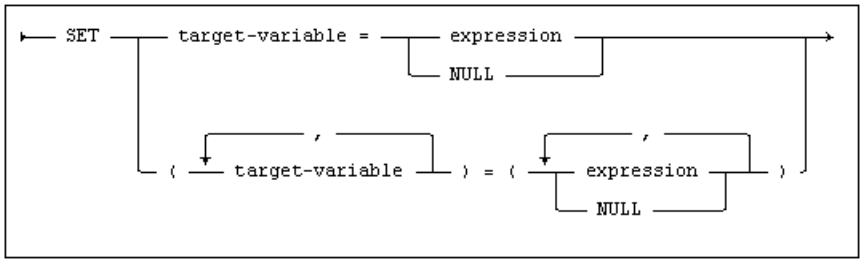
```
SELECT quantity, msi.item_id INTO :hv1, :hv2
FROM mimer_store.items AS msi
JOIN mimer_store.order_items AS msoi ON msi.item_id = msoi.item_id
ORDER BY quantity
FETCH FIRST 1 ROW ONLY;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
	Mimer SQL extension	The support for order-by clause, result-offset clause and fetch-first clause is a Mimer SQL extension.

SET

Assigns the specified value to a variable or output parameter.



Usage

Procedural, Interactive, Module, Embedded, Module.

Description

The `SET` statement directly assigns the specified `value-expression` to one or more `target-variable`'s, see *Target Variables* on page *43*. If a `target-variable` is a routine parameter, it must have mode `OUT` or `INOUT`.

Restrictions

A `value-expression` must be assignment-compatible with the data type of its `target-variable`, see *Assignments* on page *77*.

If multiple target-variables are assigned, the number of items in the row expression on the right hand of the assignment must be the same as the number of target-variables.

Notes

Where the target of the assignment is a declared variable, its name may be qualified with a scope label, see the *Mimer SQL Programmer's Manual, Chapter 11, Declaring Variables*.

If the target of the assignment is a variable declared with the `ROW` data type, a row value expression may be specified for `expression`.

If is possible to assign a value to a field of a variable declared with the ROW data type by using the following syntax to refer to the field: `routine-variable.field-name`.

See the *Mimer SQL Programmer's Manual, Chapter 11, The ROW Data Type* and *Mimer SQL Programmer's Manual, Chapter 11, Row Value Expression,* for more information.

Assignment of multiple target-variables is not supported.

Examples

```
SET firstName = 'Moira';

SET pos = position(' ' IN name);

SET book.title = 'Grapes of Wrath';
```

```
SET bookTitle = ('0-201-43328-1','JDBC API Tutorial and Reference');

SET (CITY, COUNTRY) = ('Uppsala', 'Sweden');
```

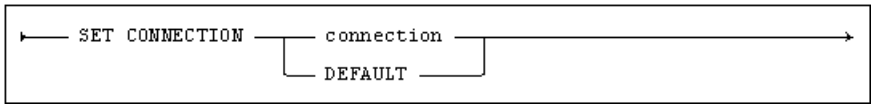
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, Assignment Using SET*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness". Feature P006, "Multiple assignment".

SET CONNECTION

Sets the current connection.



Usage

Embedded, Interactive, Module.

Description

The specified `connection` becomes current. The `connection` must specify an existing connection name. If it does not, an error code is returned and the connection status remains unchanged.

`connection` is case-sensitive.

`connection` can be specified as a host variable or a literal.

Example

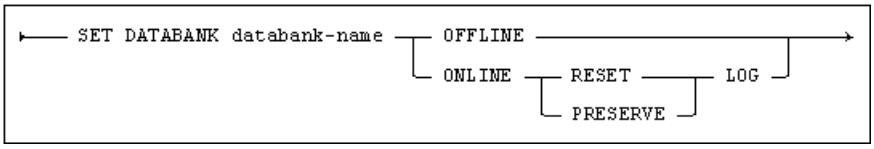
```
SET CONNECTION 'connection 2';
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F771, “Connection management”.

SET DATABANK

Sets a databank offline or online, with the option of clearing LOGDB records for it.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

Setting a databank OFFLINE makes it unavailable for all users and closes the databank file. A typical use for this is when taking databank backups using the host file system.

If the databank is being used, an error will be raised and it will not be set offline. When a databank is set offline, all online shadows will be brought up to date.

When a databank is set ONLINE again, you must specify whether to clear the LOGDB records for it (using the RESET LOG option), or whether to preserve these (using the PRESERVE LOG option). The RESET LOG option should be used after a successful backup has been taken.

It is essential to keep LOGDB in a state consistent with the databank backups, see the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data,* for a discussion of the issues.

Clearing records from LOGDB is handled automatically when CREATE BACKUP is used to take databank backups.

Restrictions

The current ident must either be the creator of the databank or have BACKUP privilege.

Notes

While a databank is OFFLINE, none of the tables stored in it are accessible and the updating of all its shadows is suspended. It is possible to use ALTER DATABANK and ALTER DATABANK RESTORE to change or recover a databank while it is OFFLINE.

If ALTER DATABANK was used to change the location of the databank file while the databank was OFFLINE, the SET DATABANK statement will verify that the new file contains a valid copy of the databank when the databank is set ONLINE again (it cannot, however, check that the contents of the file is up-to-date).

It is possible to use DROP DATABANK to drop an OFFLINE databank.

While a databank is OFFLINE, it is not possible to use ALTER SHADOW on any of its shadows.

An error will be raised if an attempt is made to set a databank OFFLINE that is already OFFLINE, or ONLINE when it is already ONLINE.

Example

```
SET DATABANK usrdb OFFLINE;
```

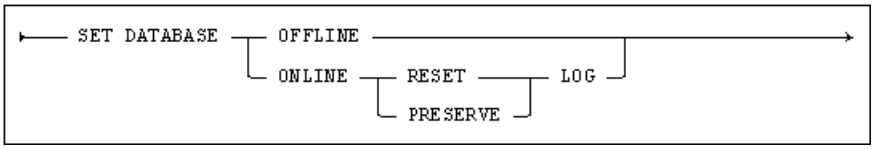
For more information, see the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The SET DATABANK statement is a Mimer SQL extension.

SET DATABASE

Sets the entire database offline or online, with the option of clearing all records from `LOGDB`.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

When the database is set `OFFLINE`, all non-system databanks are closed. During the process of setting the database `OFFLINE`, all updated databank pages are forced to disk and all databank shadows are brought up to date.

A typical use for this is when backing up all databank files in one go using the host file system utilities.

When the database is set `ONLINE` again, you must specify whether to clear all the `LOGDB` records (using the `RESET LOG` option), or whether to preserve these (using the `PRESERVE LOG` option).

The `RESET LOG` option should be used only after a complete backup has been taken of everything in the database.

Restrictions

The current ident must have `BACKUP` privilege.

Notes

While the database is `OFFLINE` no connections to it can be established, the database can only be accessed by a single system administrator ident.

An error will be raised if an attempt is made to set the database `OFFLINE` when it is already `OFFLINE`, or `ONLINE` when it is already `ONLINE`.

Example

```
SET DATABASE OFFLINE;
```

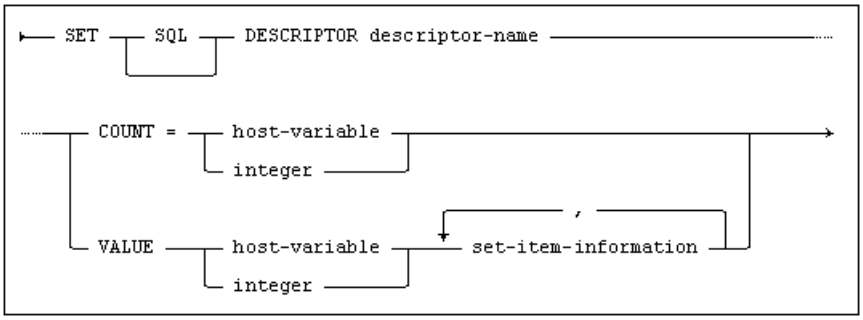
For more information, see the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data*.

Standard Compliance

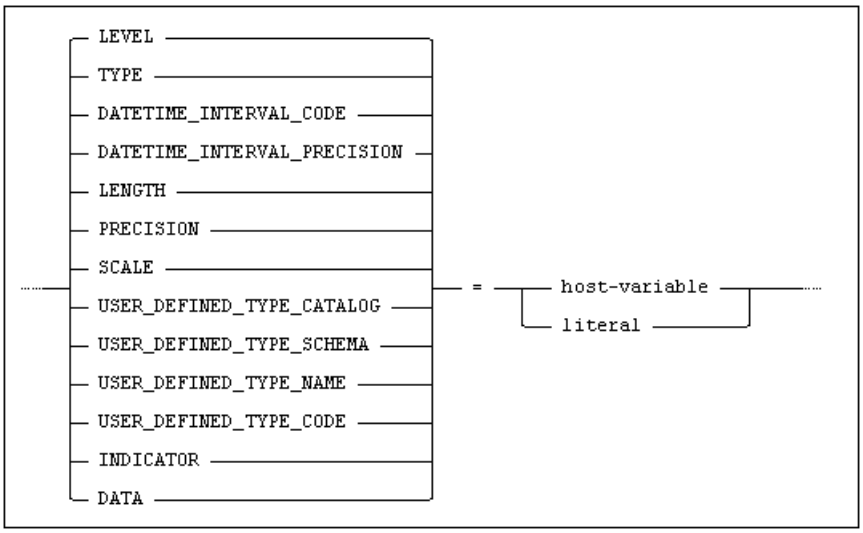
Standard	Compliance	Comments
	Mimer SQL extension	The SET DATABASE statement is a Mimer SQL extension.

SET DESCRIPTOR

Set values in an SQL descriptor area.



where `set-item-information` is:



Usage

Embedded, Module.

Description

Fields values are assigned in the specified SQL descriptor area. The `SET DESCRIPTOR` statement can be used in two forms. The `COUNT` form sets the number of active item descriptor areas for the specified SQL descriptor. The `VALUE` form assigns SQL descriptor field values for the item descriptor area specified by `item-number`.

The `descriptor-name` is identified by a host variable or a literal.

See *GET DESCRIPTOR* on page *344* for a description of the descriptor fields.

Notes

The data type of the host variables must be compatible with the data type of the associated field name.

If an item descriptor area is specified for any field other than `DATA`, the `DATA` field becomes undefined.

When the TYPE field is set, some of the other fields are implicitly set according to the table below:

Type	Implicitly set fields
BINARY	LENGTH set to 1
BINARY VARYING	LENGTH set to 1
CHARACTER	LENGTH set to 1
VARCHAR	LENGTH set to 1
NATIONAL CHARACTER	LENGTH set to 1
NCHAR VARYING	LENGTH set to 1
BLOB	LENGTH set to 1
CLOB	LENGTH set to 1
NCLOB	LENGTH set to 1
DATETIME	PRECISION set to 0
DECIMAL	PRECISION set to 15, SCALE set to 0
DOUBLE PRECISION	PRECISION set to 16
FLOAT	PRECISION set to 16
INTEGER	PRECISION set to 10
INTERVAL	DATETIME_INTERVAL_PRECISION set to 2
REAL	PRECISION set to 7
SMALLINT	PRECISION set to 5

Example

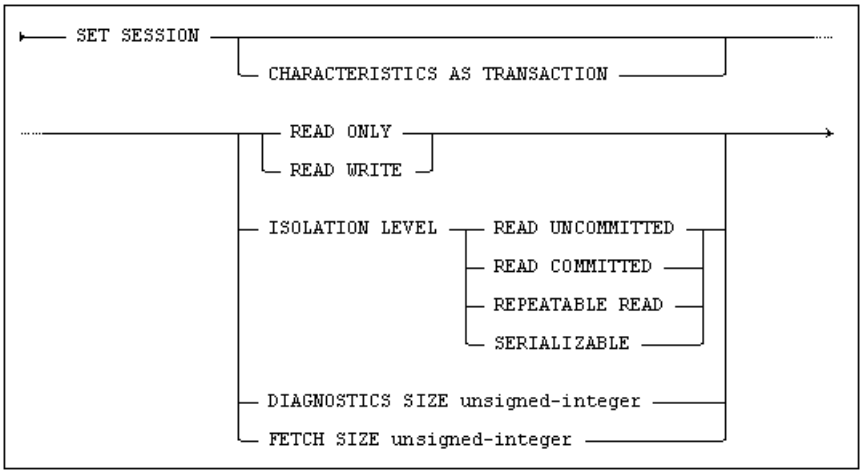
```
exec sql SET DESCRIPTOR 'descrIn' VALUE :n TYPE = :type,
                                           LENGTH = :length;
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature B031, “Basic dynamic SQL.” Feature B032, “Extended dynamic SQL” support for dynamic descriptor names.

SET SESSION

Set default mode for a session.



Usage

Embedded, Interactive, Module, Procedural.

Description

The default mode specified is set for the current connection and remains until the connection is closed.

SET SESSION READ

The `SET SESSION READ` option allows the default `SET TRANSACTION READ` setting to be defined. (The `SET TRANSACTION READ` statement only affects the single next transaction to be started after it has been used.)

The default `SET TRANSACTION READ` setting is normally `READ WRITE`, however, `SET SESSION READ` can be used to set whichever default is desired for the current session.

SET SESSION ISOLATION LEVEL

The `SET SESSION ISOLATION LEVEL` option allows the default `SET TRANSACTION ISOLATION LEVEL` setting to be defined. (The `SET TRANSACTION ISOLATION LEVEL` statement only affects the single next transaction to be started after it has been used.)

The default `SET TRANSACTION ISOLATION LEVEL` setting is normally `READ COMMITTED`, however, `SET SESSION ISOLATION LEVEL` can be used to set whichever default is desired for the current session.

If `SET SESSION ISOLATION LEVEL READ UNCOMMITTED` is specified, then a default transaction access mode of `READ ONLY` is implicit. I.e. transactions performing updates are not allowed unless a `SET TRANSACTION` statement changing this default is specified before doing such a transaction.

SET SESSION DIAGNOSTICS SIZE

The `SET SESSION DIAGNOSTICS SIZE` option allows the default size of the diagnostics area to be defined. The `unsigned-integer` value specifies how many exceptions can be stacked in the diagnostics area, and examined by `GET DIAGNOSTICS`, in situations where repeated `RESIGNAL` operations have effectively been performed. The default size is 50.

SET SESSION FETCH SIZE

The `SET SESSION FETCH SIZE` option allows for Embedded SQL (ESQL) programmers to provide hints about an appropriate block cursor size. ESQL applications will now, whenever possible, fetch result rows in blocks from the server. In effect this means that ESQL, whenever the application wants to fetch more data, transfers a number of rows from the server at once and store these in an internal buffer. Future fetches will read directly from the internal buffer until it is exhausted, when a new block of rows are requested from the server.

In most cases, this has a positive effect on performance, applications will communicate less with the server and thus improving its scalability. Communication overheads are also reduced. There are, however, a few cases when this might be detrimental to performance. One situation might be when one want the first result row as fast as possible, while there can take some time for the server to complete an entire block request. In these situations ESQL programmers may change the block fetch behavior with the session attribute `FETCH SIZE`. This attribute will provide a hint about a suitable fetch size, that is the number of rows to fetch in each block, to ESQL. ESQL, will whenever possible try to use the specified fetch size, but it may in practice use a fetch size smaller than specified. If the value is zero, the hint is ignored. The default value is zero.

Restrictions

The `SET SESSION` statement may not be issued within a transaction.

A `SET SESSION READ` setting or a `SET SESSION ISOLATION LEVEL` setting may not be changed if there are any holdable cursors remaining open from the previous transaction.

Examples

Set the default transaction isolation level to repeatable read:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Ensure that rows are transferred one at a time from the server:

```
exec sql SET SESSION FETCH SIZE 1;
```

Set the fetch block size to 24:

```
exec sql BEGIN DECLARE SECTION;
long fetch_size;
exec sql END DECLARE SECTION;
...
fetch_size = 24;
exec sql SET SESSION FETCH SIZE :fetch_size;
```

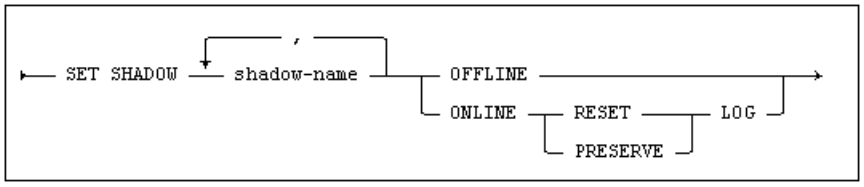
For more information, see the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	F761, “Session management”. F111, “Isolation levels other than SERIALIZABLE”.
	Mimer SQL extension	Optional CHARACTERISTICS AS TRANSACTION syntax is a Mimer SQL extension.

SET SHADOW

Sets a list of databank shadows offline or online, with the option of clearing the LOGDB records for them.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

Setting a databank shadow OFFLINE suspends updating of it. A typical use for this is when taking databank backups from shadows using the host file system.

When a databank shadow is set ONLINE again, you must specify whether:

- to clear the applicable LOGDB records using the RESET LOG option
- or whether to preserve these (using the PRESERVE LOG option).

Use the RESET LOG option after a successful backup has been taken.

Clearing records from LOGDB is handled automatically when CREATE BACKUP is used to take databank backups.

Restrictions

The current ident must either be the creator of the databank to be shadowed, or have BACKUP privilege in order to use all the SET SHADOW options.

If the current ident holds SHADOW privilege, the shadow can be set offline and online with the PRESERVE LOG option, but the RESET LOG option cannot be used.

SET SHADOW cannot be used if the master databank is OFFLINE.

Notes

While a shadow is OFFLINE, updating of it is suspended. It is possible to use ALTER SHADOW to change the shadow while it is OFFLINE.

If ALTER SHADOW was used to change the location of the shadow file while the shadow was OFFLINE, the SET SHADOW statement will verify that the new file contains a valid copy of the shadow when the shadow is set ONLINE again (it cannot, however, check that the contents of the file is up-to-date).

It is possible to use DROP SHADOW to drop an OFFLINE shadow.

SET SHADOW OFFLINE will succeed with a warning if the shadow exists and is ONLINE, but the file cannot be accessed for some reason.

It is not possible to set more than a single shadow of any given databank `OFFLINE` at a time. If the `shadow-list` contains more than one shadow for a databank, none of the shadows for that databank will be set `OFFLINE`, and an error will be raised.

An error will be raised if an attempt is made to set a shadow `OFFLINE` that is already `OFFLINE`, or `ONLINE` when it is already `ONLINE`.

Example

```
SET SHADOW MIMER_STORE$SHADOW_1, MIMER_STORE$SHADOW_2 ONLINE RESET LOG;
```

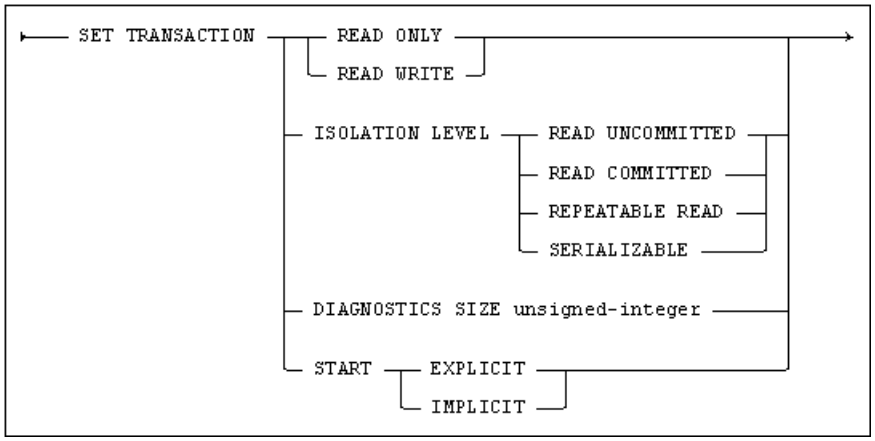
For more information, see the *Mimer SQL System Management Handbook, Chapter 10, Mimer SQL Shadowing*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The SET SHADOW statement is a Mimer SQL extension.

SET TRANSACTION

Sets transaction modes for transactions.



Usage

Embedded, Interactive, Module, Procedural.

Description

SET TRANSACTION READ

The `SET TRANSACTION READ` setting only affects the single next transaction to be started.

The default `SET TRANSACTION READ` setting (`READ WRITE` or whatever has been defined to be the default by using `SET SESSION`) applies unless an alternative is explicitly set before each transaction.

The `SET TRANSACTION READ ONLY` option is provided so that transaction performance can be optimized for those transactions not performing any updates.

It is strongly recommended that `SET TRANSACTION READ ONLY` be used for each transaction that does not require update access to the database and that `READ WRITE` mode only be used for transactions actually performing updates.

Significant performance gains can be achieved, especially for queries retrieving large numbers of rows, when the `SET TRANSACTION READ` options are used as recommended.

SET TRANSACTION ISOLATION LEVEL

The `SET TRANSACTION ISOLATION LEVEL` options are provided to control the degree to which the updates performed by a transaction are affected by the updates performed by concurrent transactions.

The `SET TRANSACTION ISOLATION LEVEL` setting only affects the single next transaction to be started.

The default `SET TRANSACTION ISOLATION LEVEL` setting (`REPEATABLE READ` or whatever has been defined to be the default by using `SET SESSION`) applies unless an alternative is explicitly set before each transaction.

If SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED is specified, then a transaction access mode of READ ONLY is implicit for the single next transaction.

All of the isolation levels guarantee that each transaction will be executed completely or not at all and that no updates will be lost.

The execution of concurrent transactions at the most secure isolation level, SERIALIZABLE, guarantees that the execution of the operations of concurrently executing transactions produces the same effect as some serial execution of those same transactions (i.e. an execution where one transaction executes to completion before the next begins).

When the other isolation levels are in effect (READ UNCOMMITTED, READ COMMITTED and REPEATABLE READ), the following effects may occur during the execution of concurrent transactions:

- 'Dirty Read' - this is where uncommitted updates can be read by another transaction. This can lead to a situation, in the event of a rollback occurring in an update transaction after another transaction has performed a read, where data has been read which (because it was never committed) must be considered to have never existed.
- 'Non-repeatable Read' - this is where a transaction reads a row and then another transaction updates or deletes that specific row. A subsequent attempt to re-read the same specific row retrieves modified information or finds that the row no longer exists, thus it can be said that the original read cannot be repeated.
- 'Phantoms' - this is where a transaction reads a set of rows that satisfy some search condition. Another transaction then performs an update which generates one or more new rows that satisfy the search condition. If the original query is repeated (using the same search condition), extra rows appear in the result set that where previously not found.

The following table summarizes, for each of the four isolation levels, which of the affects described above are guaranteed never to occur, or must be accepted as possible, where there are concurrent transactions:

Isolation Level	Dirty Read	Non-repeatable Read	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Never occurs	Possible	Possible
REPEATABLE READ	Never occurs	Never occurs	Possible
SERIALIZABLE	Never occurs	Never occurs	Never occurs

SET TRANSACTION DIAGNOSTICS SIZE

The SET TRANSACTION DIAGNOSTICS SIZE option allows the size of the diagnostics area to be defined. The unsigned-integer value specifies how many exceptions can be stacked in the diagnostics area, and examined by GET DIAGNOSTICS, in situations where repeated RESIGNAL operations have effectively been performed.

The SET TRANSACTION DIAGNOSTICS SIZE setting only affects the single next transaction to be started.

The default SET TRANSACTION DIAGNOSTICS SIZE setting (50 or whatever has been defined to be the default by using SET SESSION) applies unless an alternative is explicitly set before each transaction.

SET TRANSACTION START

Transactions are started either by an explicit START statement or by an implicit transaction start. The procedure that is followed is determined by using the SET TRANSACTION START statement.

When START is set to IMPLICIT, the first operation involving a databank with either the TRANSACTION or LOG option will start a transaction. The transaction must then be terminated explicitly by either COMMIT or ROLLBACK.

The SET TRANSACTION START setting has effect in the current session until SET TRANSACTION START is next used.

The default setting is START IMPLICIT.

Restrictions

The SET TRANSACTION statement may not be issued within a transaction.

A SET TRANSACTION READ setting or a SET TRANSACTION ISOLATION LEVEL setting may not be changed if there are any holdable cursors remaining open from the previous transaction.

Notes

The SET TRANSACTION START statement is generally issued at the beginning of a session, to set the start mode for transactions. Changing the start mode for transactions in the middle of a session is not generally recommended.

The SET SESSION statement can be used to define the default settings for the SET TRANSACTION READ, SET TRANSACTION ISOLATION LEVEL and SET TRANSACTION DIAGNOSTICS SIZE options.

Example

```
exec sql SET TRANSACTION START EXPLICIT;
LOOP
   exec sql FETCH C1 INTO :var1,:var2,...,:varn;
   display var1,var2,...,varn;
   prompt "Update row?";
   exit when answer = "yes";
END LOOP
exec sql START;
exec sql UPDATE table SET ...
         WHERE col1 = :var1,
               col2 = :var2, ...;
exec sql COMMIT;
```

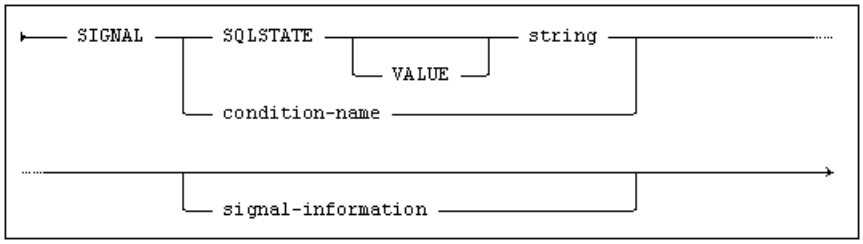
Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

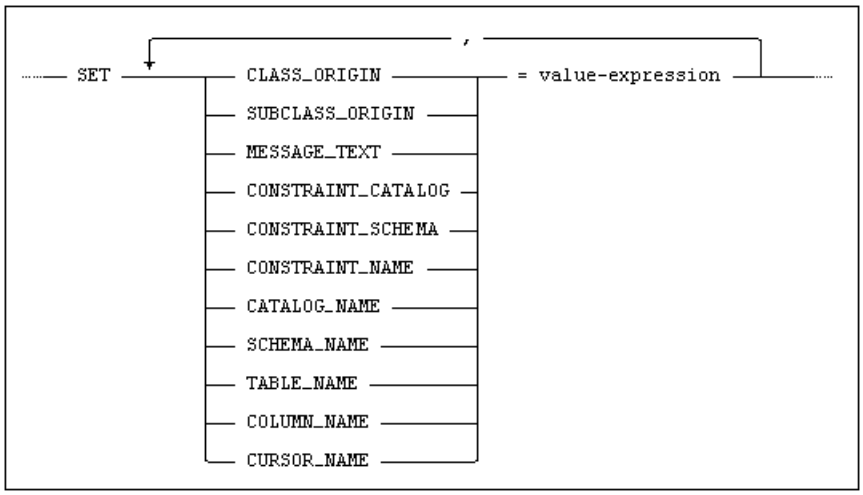
Standard	Compliance	Comments
SQL-2016	Features outside core	Feature F111, “Isolation levels other than serializable” support for READ UNCOMMITTED, READ COMMITTED and REPEATABLE READ. Feature F121, “Basic diagnostics management” support for DIAGNOSTICS SIZE.
	Mimer SQL extension	The form TRANSACTION START is a Mimer SQL extension.

SIGNAL

Raises the specified exception condition.



where *signal-information* is:



Usage

Procedural.

Description

The `SIGNAL` statement has the effect of raising an exception condition specified by an `SQLSTATE` value or a condition name.

If a condition identifier is used in the statement the associated SQLSTATE value is raised. If the condition identifier is declared without an `SQLSTATE` value, the `SQLSTATE 45000` is raised. If there is an appropriate exception handler for this `SQLSTATE`, this handler will be invoked otherwise the `SQLSTATE` is propagated to the calling environment.

It is possible to provide diagnostics information with the signal statement. This diagnostics information can be retrieved where the exception is handled. This can be used to customize error messages for an application. The signal-information fields are described in the GET DIAGNOSTICS section *condition-info Information Items* on page *353*.

Note that an `SQLSTATE` value corresponding to a warning condition that is not catched by an exception handler will not be propagated to the calling environment.

Notes

See *DECLARE CONDITION* on page *307* for a description of how to declare a condition name.

Example

```
signal sqlstate 'UE324'
   set message_text = 'A horse named ' || horse_name ||
                      ' already exists in the database';
```

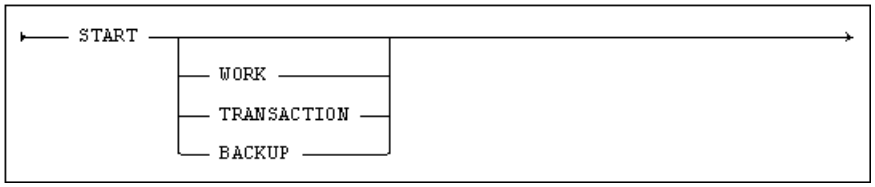
For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, SIGNAL Statements*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, "Computational completeness".

START

Starts a transaction build-up.



Usage

Embedded, Interactive, Module, Procedural.

Description

A new transaction is started, regardless of whether the transaction start mode is set to `IMPLICIT` or `EXPLICIT`, see the *SET TRANSACTION* on page *418* statement.

The `START BACKUP` command starts a transaction in which the `CREATE ONLINE BACKUP` statements of an online backup sequence are executed, see the description of *CREATE BACKUP* on page *248* for more information.

Restrictions

The `START` statement may not be executed from within a transaction.

The `START` statement may not be used in a result set procedure.

The `START BACKUP` command is not supported in procedural mode.

Example

```
exec sql SET TRANSACTION START EXPLICIT;

LOOP
   exec sql FETCH C1 INTO :var1,:var2,...,:varn;
   DISPLAY var1,var2,...,varn;
   PROMPT "Update row?";
   EXIT WHEN ANSWER = "yes";
END LOOP

exec sql START;
exec sql UPDATE table SET ...
         WHERE col1 = :var1,
               col2 = :var2, ...
exec sql COMMIT;
```

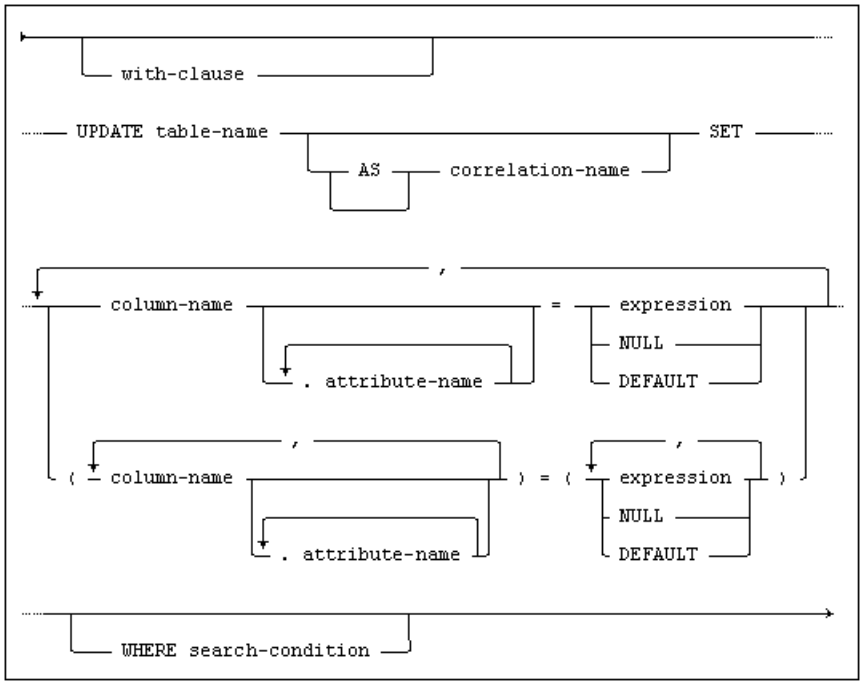
For more information, see the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature T241, “START TRANSACTION statement”.
	Mimer SQL extension	The use of the keywords BACKUP and WORK is a Mimer SQL extension.

UPDATE

Updates a set of rows in a table or view.



Usage

Embedded, Interactive, Module, Procedural, ODBC, JDBC.

Description

The table or view identified by the table name is updated in the rows which satisfy the condition in the `WHERE` clause by assigning new values to the columns as specified in the `SET` clause. If no `WHERE` clause is specified, all rows are updated.

Values to be assigned to columns may be specified either as expressions or by using the keywords `NULL` or `DEFAULT`. Expressions must have a data type compatible with the definition of the column to which they are assigned. If column names are used in expressions, they must refer to columns in the table or view addressed in the `UPDATE` clause. The value specified by a column name in an expression is the value for the column in the row concerned before any update operation is performed.

If no row is updated a `NOT FOUND` condition code is returned, see *Appendix E Return Status and Conditions*.

Language Elements

`expression`, see *Chapter 9, Expressions and Predicates*.

`search-condition`, see *Chapter 10, Search Conditions*.

`with-clause`, see *The WITH Clause* on page *179*.

Restrictions

UPDATE access is required on the columns specified in the SET clause.

If the UPDATE statement is used on a primary key column of a table, the table must be stored in a databank with the TRANSACTION or LOG option.

In a procedural usage context, the UPDATE statement is only permitted if the procedure access-clause is MODIFIES SQL DATA, see *CREATE PROCEDURE* on page *271*.

Notes

Column names on the left-hand side of the assignment operator in the SET clause may not be qualified by the table reference.

Columns may not be specified more than once on the left-hand side of the assignment operator in the SET clause in a single UPDATE statement.

Expressions used in the SET clause cannot refer to set functions (except for in a subquery).

Column names in the search condition of the WHERE clause must identify columns in the table or view to be updated.

If a correlation name is introduced after the table reference in the UPDATE clause, the correlation name must be used to refer to the table in the WHERE clause of the same UPDATE statement.

UNIQUE and CHECK constraints in the table being updated may not be violated (this is evaluated at the end when all the modifications involved in the UPDATE statement have been made).

If the table name specified in the UPDATE statement is subject to any referential constraint, the values in all updated rows must conform to that constraint. If a view defined WITH CHECK OPTION is to be updated, the values assigned to the columns must conform to the view definition.

Read-only views may not be updated, see *CREATE VIEW* on page *302*.

An UPDATE statement is executed as a single statement. If an error occurs at any point during the execution, no rows will be updated (however, if the table is stored in a databank with the WORK option it is possible that some rows will be updated).

Example

The following example is taken from the *Mimer SQL User's Manual, Chapter 5, Updating Tables*.

```
UPDATE currencies SET exchange_rate = 7.25 WHERE currency_code = 'USD';
```

Multiple column update example:

```
UPDATE currencies
SET exchange_rate = 36.38, currency = 'Jaimacan Dollars'
WHERE currency_code = 'JMD';
```

can also be written as:

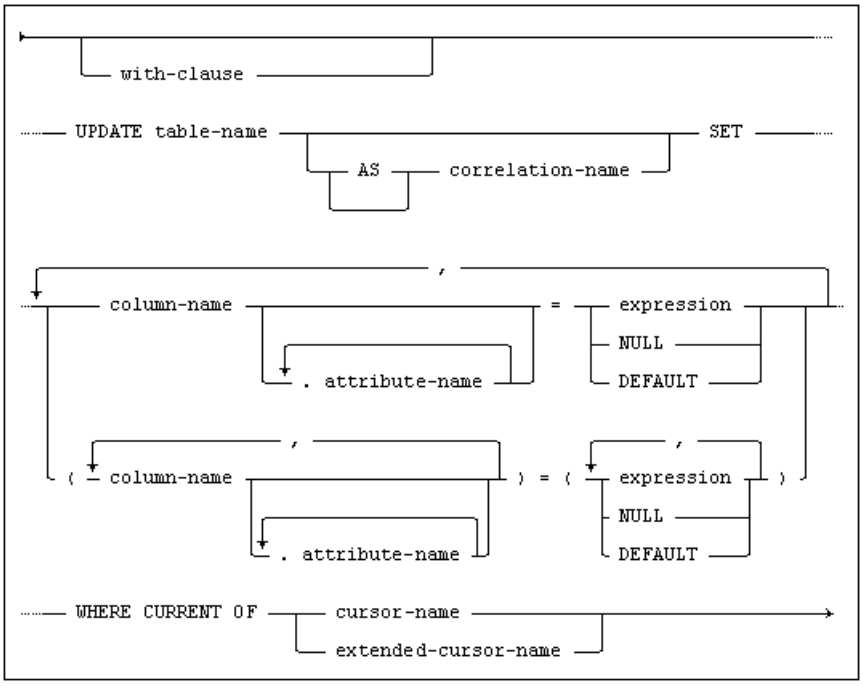
```
UPDATE currencies
SET (exchange_rate, currency) = (36.38, 'Jaimacan Dollars')
WHERE currency_code = 'JMD';
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F781, “Self-referencing operations” the update table can be used in search conditions in the update statement.

UPDATE CURRENT

Updates the current row indicated by a cursor.



Usage

Embedded, Module, ODBC, Procedural, JDBC.

Description

The current row addressed by the cursor is updated by assigning new values to the columns as specified in the SET clause.

See *ALLOCATE CURSOR* on page *196* for a description of extended cursors.

If an extended cursor is used in an `UPDATE CURRENT` statement, the cursor is represented following these rules:

- If the `UPDATE CURRENT` statement is executed with static SQL, i.e. using `EXEC SQL`, the extended cursor is represented by the host variable containing the cursor.
- If the `UPDATE CURRENT` statement is executed with dynamic SQL, the extended cursor must be represented by the cursor value contained in the host variable.

Values to be assigned to columns may be specified either as expressions or by using the keywords `NULL` or `DEFAULT`. Expressions must have a data type compatible with the definition of the column to which they are assigned.

If column names are used in expressions, they must refer to columns in the table or view addressed in the `UPDATE CURRENT` clause. The value specified by a column name in an expression is the value for the column in the row concerned before the update operation is performed.

Language Elements

`expression`, see *Chapter 9, Expressions and Predicates*.

`with-clause`, see *The WITH Clause* on page *179*.

Restrictions

`UPDATE` access to the appropriate columns in the table or view identified by the table name is required when the cursor used for the `UPDATE CURRENT` statement is opened. If `UPDATE` access is not held, the cursor may be opened but `UPDATE CURRENT` statements will fail. Direct access to the base table is not required for an update operation on a view.

If the `UPDATE CURRENT` statement is used on a primary key column of a table, the table must be stored in a databank with the `TRANSACTION` or `LOG` option.

In a procedural usage context, `extended-cursor-name` cannot be used to identify the cursor.

In a procedural usage context, the `UPDATE CURRENT` statement is only permitted if the procedure `access-clause` is `MODIFIES SQL DATA`, see *CREATE PROCEDURE* on page *271*.

A row indicated by a `WITH HOLD` cursor must have been fetched in the same transaction.

Notes

Column names on the left-hand side of the assignment operator in the `SET` clause may not be qualified by the table name.

Columns may not be specified more than once on the left-hand side of the assignment operator in the `SET` clause in a single `UPDATE` statement.

Expressions used in the `SET` clause cannot refer to set functions (except for in a subquery).

If columns are listed in the `FOR UPDATE OF` clause of the cursor definition (described under `SELECT`) no other columns may be specified on the left-hand side of the assignment operators in the `SET` clause.

The table name specified in the `UPDATE CURRENT` clause must be exactly the same as that in the `FROM` clause of the `SELECT` statement used to declare the cursor. If a synonym is used in one of the statements, the same synonym must also be used in the other.

`UNIQUE` constraints in the table being updated may not be violated.

If the table name specified in the `UPDATE` statement is subject to any referential constraint, the values in the row to be updated must conform to that constraint.

If a view defined `WITH CHECK OPTION` is to be updated, the values assigned to the columns must conform to the view definition.

The `UPDATE CURRENT` statement may not be used for read-only cursors.

Example

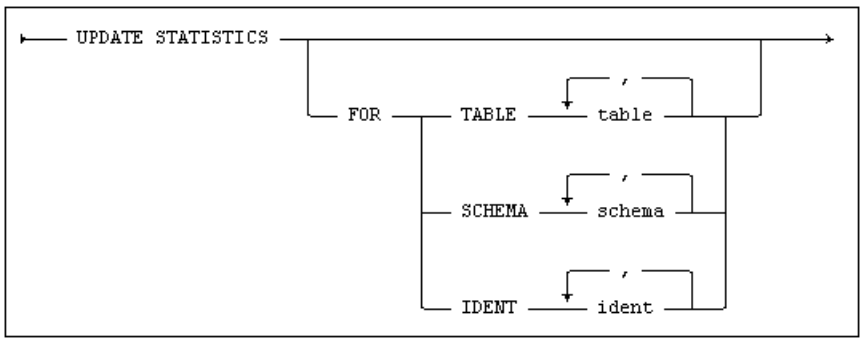
```
...
FETCH C1 INTO I_CHARGE_CODE,I_AMOUNT;
    IF I_CHARGE_CODE = '270' AND ... THEN
        UPDATE BILL SET AMOUNT = AMOUNT * 1.10 WHERE CURRENT OF C1;
...
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F831, “Full cursor update”.

UPDATE STATISTICS

Updates the statistics recorded for all tables in the database, a specified list of tables, all tables in a specified list of schemas or all the tables belonging to the schemas owned by a specified list of idents.



Usage

Embedded, Interactive, Module, ODBC, JDBC.

Description

The default operation is to update statistics for all tables, including data dictionary tables, in the database.

It is possible to update statistics for a specified list of tables by using the `FOR TABLE` option, for all tables belonging to a specified list of schemas by using the `FOR SCHEMA` option, or for all the tables belonging to the schemas created by a specified list of idents by using the `FOR IDENT` option.

Update statistics includes an automatic operation which ensures the consistency of secondary indexes (both explicitly created indexes and those created by the system when certain constraints are defined). The operation is transparent to users of the database and is performed on indexes selected by the `UPDATE STATISTICS` statement that are contained in a databank with the `TRANSACTION` or `LOG` option and which are flagged as ‘not consistent’.

The process of ensuring the consistency of an index, and updating statistics for all tables in the database (the default operation), can be rather time-consuming. Therefore, it is generally recommended that these operations be performed at off-peak times, refer to the *Mimer SQL System Management Handbook, Chapter 11, Database Statistics* for more information.

A secondary index is flagged as `not consistent` if it belongs to a table in a databank with the `WORK` option, or if the databank containing it has been upgraded from Mimer SQL version 8.1 (or older).

The `IS_CONSISTENT` column in the data dictionary table `TABLE_CONSTRAINTS` shows which indexes in the database are flagged as `not consistent`.

Restrictions

The current ident must be the creator of all the tables involved or must have `STATISTICS` privilege.

Notes

The `UPDATE STATISTICS` statement can be used concurrently with other SQL statements.

Precompiled statements may change search orders depending on the result of the updated statistics.

Example

```
UPDATE STATISTICS FOR IDENT joe;
```

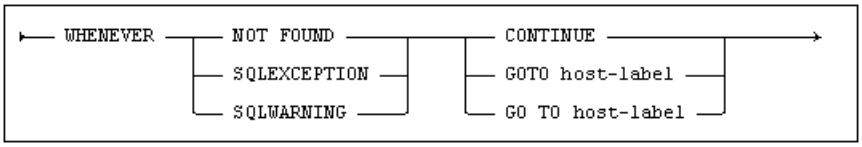
For more information, see the *Mimer SQL System Management Handbook, Chapter 11, Database Statistics*.

Standard Compliance

Standard	Compliance	Comments
	Mimer SQL extension	The UPDATE STATISTICS statement is a Mimer SQL extension.

WHENEVER

Defines action to be taken for errors and exception conditions.



Usage

Embedded.

Description

The action taken in the event of a condition arising during execution of an SQL statement is governed by the most recently issued `WHENEVER` statement. There are three different types of conditions: `NOT FOUND`, `SQLEXCEPTION` and `SQLWARNING`. See *Appendix E Return Status and Conditions* for a description of the different condition types.

The action taken is as follows:

- `CONTINUE`

  Program execution continues at the next sequential statement of the source program.

- `GOTO`

  Program execution continues at the source code statement identified by `host-label`, where `host-label` is a program label in a program written according to the host language.

Notes

If a condition in an SQL statement is not covered by an explicit `WHENEVER` statement issued earlier in the host program code, `CONTINUE` will be assumed.

It's recommended to set `SQLEXCEPTION` to `CONTINUE` action first of all in the diagnostics part of the code, to avoid the risk of a looping application.

Example

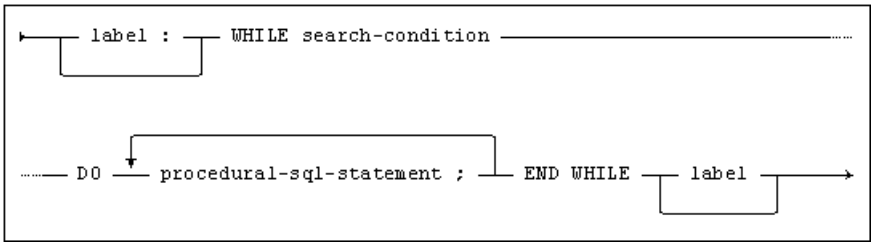
```
      ...
      GOTO 1025
      EXEC SQL WHENEVER SQLEXCEPTION GOTO 1600
1025  CONTINUE
      EXEC SQL DELETE FROM MYTABLE
      ...
1060  CONTINUE
      EXEC SQL WHENEVER SQLEXCEPTION CONTINUE
      EXEC SQL GET DIAGNOSTICS ...
```

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.

WHILE

Allows one or more procedural SQL statements to be iteratively executed.



Usage

Procedural.

Description

The `WHILE` statement can be used to iteratively execute a sequence of one or more `procedural-sql-statements`.

The iteration continues as long as `search-condition` evaluates to true.

For information on `procedural-sql-statements`, see *Procedural SQL Statements* on page *193*.

Restrictions

If `label` appears at the beginning and at the end of the `WHILE` statement, the same value must be specified in both places.

Specifying `label` is optional, however, if `label` appears at the end of the `WHILE` statement, it must also appear at the beginning.

A label is required at the beginning if the `LEAVE` statement is to be used to terminate the `WHILE` statement.

Notes

The `WHILE` statement may be terminated by executing the `LEAVE` statement using `label`. It will also terminate if an exception condition is raised, in accordance with the normal exception handling process.

Example

```
SET I = 0;
L1:
WHILE I <= 10 DO
  ...
  SET I = I + 1;
END WHILE L1;
```

For more information, see the *Mimer SQL Programmer's Manual, Chapter 11, Iteration Using WHILE*.

Standard Compliance

Standard	Compliance	Comments
SQL-2016	Features outside core	Feature P002, “Computational completeness”.

Chapter 13

Data Dictionary Views

This chapter documents the predefined system views on the data dictionary tables. PUBLIC holds SELECT access on these views, so that they may be examined by any user.

INFORMATION_SCHEMA views are used to retrieve information about the objects in the data dictionary.

An INFORMATION_SCHEMA view can be read with the statement (note the qualified form of view-name):

```
SELECT column-list
FROM INFORMATION_SCHEMA.view-name
WHERE condition
```

Many of the views include only objects, privileges and so on relevant to the current ident (the description for each view indicates exactly what information is shown in the view).

Note: Some of the views have columns designed to display information that is not currently supported by Mimer SQL (e.g. catalog names for database objects), in this situation the empty string ("") will be shown in these columns.

The tables in the data dictionary may be read directly only by the system administrator ident SYSADM in the default installation. The base tables in the data dictionary are documented in the *Mimer SQL System Management Handbook*. The system administrator may, if desired, grant SELECT access on the dictionary tables to other users.

No user may access the data dictionary views or tables directly for any purpose other than SELECT. All data dictionary maintenance is performed by internal routines and is invisible to the user.

INFORMATION_SCHEMA dictionary views

The table below summarizes the data dictionary views that are part of the schema INFORMATION_SCHEMA:

View name	Description
INFORMATION_SCHEMA.ASSERTIONS	Owned assertions.
INFORMATION_SCHEMA.ATTRIBUTES	Owned user-defined type attributes.

View name	Description
INFORMATION_SCHEMA.CHARACTER_SETS	Accessible character sets.
INFORMATION_SCHEMA.CHECK_CONSTRAINTS	Owned check constraints.
INFORMATION_SCHEMA.COLLATIONS	Accessible collations.
INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE	Columns defined using owned domains.
INFORMATION_SCHEMA.COLUMN_PRIVILEGES	Privileges granted on accessible table columns.
INFORMATION_SCHEMA.COLUMN_UDT_USAGE	Columns defined using owned user-defined types.
INFORMATION_SCHEMA.COLUMNS	Accessible table columns.
INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE	Columns referenced by owned referential, unique, check or assertion constraints.
INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE	Tables on which owned referential, unique, check or assertion constraints are defined.
INFORMATION_SCHEMA.DIRECT_SUPERTABLES	Information about inheritance relations between tables.

View name	Description
INFORMATION_SCHEMA.DIRECT_SUPERTYPES	Information about inheritance relations between user-defined types.
INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS	Constraints of accessible domains.
INFORMATION_SCHEMA.DOMAINS	Accessible domains.
INFORMATION_SCHEMA.EXT_ACCESS_PATHS	All explicit and implicit indexes for tables.
INFORMATION_SCHEMA.EXT_COLLATION_DEFINITIONS	Collation definitions.
INFORMATION_SCHEMA.EXT_COLUMN_OFFSET_INFORMATION	Physical structure of columns in a table.
INFORMATION_SCHEMA.EXT_COLUMN_REMARKS	Comments for accessible table columns.
INFORMATION_SCHEMA.EXT_DATABANKS	Accessible databanks.
INFORMATION_SCHEMA.EXT_IDENTS	Accessible authorization idents.
INFORMATION_SCHEMA.EXT_INDEX_COLUMN_USAGE	Accessible table columns on which indexes depend.
INFORMATION_SCHEMA.EXT_INDEXES	Accessible indexes.
INFORMATION_SCHEMA.EXT_OBJECT_IDENT_USAGE	Accessible objects created by authorization ident.

View name	Description
INFORMATION_SCHEMA.EXT_OBJECT_OBJECT_USED	Accessible objects used by other objects.
INFORMATION_SCHEMA.EXT_OBJECT_OBJECT_USING	Accessible objects using other objects.
INFORMATION_SCHEMA.EXT_OBJECT_PRIVILEGES	Object privileges granted to an authorization ident.
INFORMATION_SCHEMA.EXT_ONEROW	Dummy view with one single row.
INFORMATION_SCHEMA.EXT_ROUTINE_MODULE_DEFINITION	Source definition for routines defined in modules.
INFORMATION_SCHEMA.EXT_ROUTINE_MODULE_USAGE	Accessible routines in a module.
INFORMATION_SCHEMA.EXT_SCHEMAS	Schemas containing objects to which current user have some access.
INFORMATION_SCHEMA.EXT_SEQUENCES	Accessible sequences.
INFORMATION_SCHEMA.EXT_SHADOWS	Accessible shadows.
INFORMATION_SCHEMA.EXT_SOURCE_DEFINITION	Text definition for owned objects.

View name	Description
INFORMATION_SCHEMA.EXT_STATEMENT_DEFINITION	Shows a textual definition of the precompiled statements available to the current ident.
INFORMATION_SCHEMA.EXT_STATEMENTS	Shows all precompiled statements available to the current ident.
INFORMATION_SCHEMA.EXT_STATISTICS	Statistics for table.
INFORMATION_SCHEMA.EXT_SYNONYMS	Accessible synonyms.
INFORMATION_SCHEMA.EXT_SYSTEM_PRIVILEGES	System privileges granted to an authorization ident.
INFORMATION_SCHEMA.EXT_TABLE_DATABANK_USAGE	Owned databanks on which tables depend.
INFORMATION_SCHEMA.KEY_COLUMN_USAGE	Table columns constrained as keys by accessible tables.
INFORMATION_SCHEMA.METHOD_SPECIFICATION_PARAMETERS	Accessible method's parameters.
INFORMATION_SCHEMA.METHOD_SPECIFICATIONS	Accessible method specifications.
INFORMATION_SCHEMA.MODULES	Owned modules.

View name	Description
INFORMATION_SCHEMA.PARAMETERS	Parameters of accessible routines.
INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS	Accessible tables' referential constraints.
INFORMATION_SCHEMA.ROUTINE_COLUMN_USAGE	Owned table columns on which routines depend.
INFORMATION_SCHEMA.ROUTINE_PRIVILEGES	Privileges held on accessible routines.
INFORMATION_SCHEMA.ROUTINE_TABLE_USAGE	Owned tables on which routines depend.
INFORMATION_SCHEMA.ROUTINES	Accessible routines.
INFORMATION_SCHEMA.SCHEMATA	Owned schemas.
INFORMATION_SCHEMA.SEQUENCES	Accessible sequences.
INFORMATION_SCHEMA.SQL_FEATURES	Features and subfeatures of SQL-2016.
INFORMATION_SCHEMA.SQL_LANGUAGES	Conformance levels for supported SQL language options and dialects.
INFORMATION_SCHEMA.SQL_SIZING	Sizing items.
INFORMATION_SCHEMA.TABLE_CONSTRAINTS	Accessible tables' constraints.
INFORMATION_SCHEMA.TABLE_PRIVILEGES	Privileges held on accessible tables.

View name	Description
INFORMATION_SCHEMA.TABLES	Accessible tables.
INFORMATION_SCHEMA.TRANSLATIONS	Accessible character set translations.
INFORMATION_SCHEMA.TRIGGER_COLUMN_USAGE	Owned columns referenced from a trigger action.
INFORMATION_SCHEMA.TRIGGER_TABLE_USAGE	Tables on which owned triggers depend.
INFORMATION_SCHEMA.TRIGGERED_UPDATE_COLUMNS	Owned columns referenced from UPDATE trigger column lists.
INFORMATION_SCHEMA.TRIGGERS	Owned triggers.
INFORMATION_SCHEMA.UDT_PRIVILEGES	Privileges for accessible user-defined types.
INFORMATION_SCHEMA.USAGE_PRIVILEGES	USAGE privilege held on accessible objects.
INFORMATION_SCHEMA.USER_DEFINED_TYPES	Owned user-defined types.
INFORMATION_SCHEMA.VIEW_COLUMN_USAGE	Columns on which owned views depend.
INFORMATION_SCHEMA.VIEW_TABLE_USAGE	Tables on which owned views depend.
INFORMATION_SCHEMA.VIEWS	Accessible views.

INFORMATION_SCHEMA.ASSERTIONS

The ASSERTIONS system view shows all assertions owned by the current ident.

Column name	Data type	Description
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the assertion.
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the assertion.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the assertion.
IS_DEFERRABLE	VARCHAR(3)	One of: YES = the assertion is deferrable NO = the assertion is not deferrable
INITIALLY_DEFERRED	VARCHAR(3)	One of: YES = the assertion is immediate NO = the assertion is deferred.

INFORMATION_SCHEMA.ATTRIBUTES

Contains one row for each attribute of a user-defined type accessible to the current user

Column name	Data type	Description
UDT_CATALOG	NCHAR VARYING(128)	Name of catalog containing user-defined type.
UDT_SCHEMA	NCHAR VARYING(128)	Name of schema containing user-defined type
UDT_NAME	NCHAR VARYING(128)	Name of user-defined type
ATTRIBUTE_NAME	NCHAR VARYING(128)	Name of attribute
ORDINAL_POSITION	INTEGER	Ordinal position for attribute within user-defined type
ATTRIBUTE_DEFAULT	NCHAR VARYING(400)	Default value for attribute
IS_NULLABLE	VARCHAR(3)	Nullability attribute YES = The attribute can be null NO = The attribute can not be null

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	The type of the attribute. One of: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHARACTER LARGE OBJECT NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED
CHARACETR_MAXIMUM_LENGTH	INTEGER	For a character data type, this shows the maximum length in characters. For all other data types it is the null value.
CHARACTER_OCTET_LENGTH	INTEGER	For a character data type, this shows the maximum length in octets. For all other data types it is the null value. It the same value as CHARACTER_MAXIMUM length for single octet data types.
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of significant digits allowed in the column. For all other data types it is the null value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.

Column name	Data type	Description
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier. Can be one of: YEAR YEAR TO MONTH DAY HOUR MINUTE SECOND DAY TO HOUR DAY TO MINUTE DAY TO SECOND HOUR TO MINUTE HOUR TO SECOND MINUTE TO SECOND. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision (see *Interval Literals* on page *67*). For other data types it is the null value.
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set used by the attribute
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set used by the attribute
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set used by the attribute
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation used by the attribute.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation used by the attribute.

Column name	Data type	Description
COLLATION_NAME	NCHAR VARYING(128)	The name of the collation used by the attribute.
USER_DEFINED_TYPE_CATALOG	NCHAR VARYING(128)	The name of the user-defined type catalog used by the attribute.
USER_DEFINED_TYPE_SCHEMA	NCHAR VARYING(128)	The name of the user-defined type schema used by the attribute.
USER_DEFINED_TYPE_NAME	NCHAR VARYING(128)	The name of the user-defined type name

INFORMATION_SCHEMA.CHARACTER_SETS

The CHARACTER_SETS system view describes each character set to which the current ident has USAGE privilege.

Column name	Data type	Description
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set.
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set.
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set.
FORM_OF_USE	NCHAR VARYING(128)	A user-defined name that indicates the form-of-use of the character set.
NUMBER_OF_CHARACTERS	INTEGER	The number of characters in the character set.
DEFAULT_COLLATE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the default collation for the character set.
DEFAULT_COLLATE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the default collation for the character set.
DEFAULT_COLLATE_NAME	NCHAR VARYING(128)	The name of the default collation for the character set.

INFORMATION_SCHEMA.CHECK_CONSTRAINTS

The CHECK_CONSTRAINTS system view lists the check constraints that are owned by the current ident.

Column name	Data type	Description
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the check constraint.
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the check constraint.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the check constraint.
CHECK_CLAUSE	NCHAR VARYING(200)	The character representation of the search condition used in the check clause. If the character representation does not fit, the value is null. In that case the definition can be found in INFORMATION_SCHEMA.EXT_SOURCE_DEFINITION.

INFORMATION_SCHEMA.COLLATIONS

The COLLATIONS system view describes each collation to which the current ident has access.

Column name	Data type	Description
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation.
COLLATION_NAME	NCHAR VARYING(128)	Name of the collation.
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set on which the collation is defined.
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set on which the collation is defined.
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set on which the collation is defined.

Column name	Data type	Description
PAD_ATTRIBUTE	VARCHAR(20)	One of the following values: NO PAD = the collation has the no pad attribute PAD SPACE = the collation has the pad space attribute.

INFORMATION_SCHEMA.COLUMN_DOMAIN_USAGE

The COLUMN_DOMAIN_USAGE system view lists the table columns which depend on domains owned by the current ident.

Column name	Data type	Description
DOMAIN_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the domain.
DOMAIN_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the domain.
DOMAIN_NAME	NCHAR VARYING(128)	The name of the domain.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
COLUMN_NAME	NCHAR VARYING(128)	The name of the column.

INFORMATION_SCHEMA.COLUMN_PRIVILEGES

The COLUMN_PRIVILEGES system view lists privileges on table columns that were granted by the current ident and privileges on table columns that were granted to the current ident or to PUBLIC.

Column name	Data type	Description
GRANTOR	NCHAR VARYING(128)	The name of the ident who granted the privilege.
GRANTEE	NCHAR VARYING(128)	The name of the ident to whom the privilege was granted. Granting a privilege to PUBLIC will result in only one row (per privilege granted) in this view and the name PUBLIC will be shown.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table containing the column on which the column privilege has been granted.
COLUMN_NAME	NCHAR VARYING(128)	The name of the column on which the privilege has been granted.
PRIVILEGE_TYPE	VARCHAR(20)	A value describing the type of the column privilege that was granted. One of: INSERT REFERENCES SELECT UPDATE. Note that when multiple table column privileges are granted to the same user at the same time (e.g. when the keyword ALL is used), multiple rows appear in this view (one for each privilege granted).
IS_GRANTABLE	VARCHAR(3)	One of: YES = the privilege is held WITH GRANT OPTION NO = the privilege is not held WITH GRANT OPTION.

INFORMATION_SCHEMA.COLUMN_UDT_USAGE

Contains one row for each column using a user-defined type created by the current user.

Column name	Data type	Description
UDT_CATALOG	NCHAR VARYING(128)	Name of catalog containing user-defined type.
UDT_SCHEMA	NCHAR VARYING(128)	Name of schema containing user-defined type
UDT_NAME	NCHAR VARYING(128)	Name of user-defined type
TABLE_CATALOG	NCHAR VARYING(128)	Catalog name for table using UDT
TABLE_SCHEMA	NCHAR VARYING(128)	Schema name for table using UDT
TABLE_NAME	NCHAR VARYING(128)	Name of table using UDT

Column name	Data type	Description
COLUMN_NAME	NCHAR VARYING(128)	Name of column using UDT

INFORMATION_SCHEMA.COLUMNS

The COLUMNS system view lists the table columns to which the current ident has access.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table or view.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view.
TABLE_NAME	NCHAR VARYING(128)	The name of the table or view.
COLUMN_NAME	NCHAR VARYING(128)	The name of the column of the table or view.
ORDINAL_POSITION	INTEGER	The ordinal position of the column in the table. The first column in the table is number 1.

Column name	Data type	Description
COLUMN_DEFAULT	NCHAR VARYING(200)	This shows the default value for the column. If the default value is a character string, the value shown is the string enclosed in single quotes. If the default value is a numeric literal, the value is shown in its original character representation without enclosing quotes. If the default value is a DATE, TIME or TIMESTAMP, the value shown is the appropriate keyword (e.g. DATE) followed by the literal representation of the value enclosed in single quotes (see *DATE, TIME and TIMESTAMP Literals* on page *67* for a description of DATE, TIME and TIMESTAMP literals). If the default value is a pseudo-literal, the value shown is the appropriate keyword (e.g. CURRENT_DATE) without enclosing quotes. If the default value is the null value, the value shown is the keyword NULL without enclosing quotes. If the default value cannot be represented without truncation, then TRUNCATED is shown without enclosing quotes. If no default value was specified then its value is the null value. The value of COLUMN_DEFAULT is syntactically suitable for use in specifying default-value in a CREATE TABLE or ALTER TABLE statement (except when TRUNCATED is shown).
IS_NULLABLE	VARCHAR(3)	One of: NO = the column is not nullable, according to the rules in the international standard YES = the null value is allowed in the column.

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	Identifies the data type of the column. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
LOB_MAXIMUM_LENGTH	BIGINT	For the LOB data type, this shows the maximum length in bytes. For all other data types it is the null value.
CHARACTER_MAXIMUM_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in characters or bytes. For all other data types it is the null value.
CHARACTER_OCTET_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in octets. For all other data types it is the null value. For single octet character sets, this is the same as CHARACTER_MAX_LENGTH.
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of significant digits allowed in the column. For all other data types it is the null value.

Column name	Data type	Description
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier. Can be one of: YEAR YEAR TO MONTH DAY HOUR MINUTE SECOND DAY TO HOUR DAY TO MINUTE DAY TO SECOND HOUR TO MINUTE HOUR TO SECOND MINUTE TO SECOND. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision (see *Interval Qualifiers* on page *55*). For other data types it is the null value.
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set used by the column.
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set used by the column.

Column name	Data type	Description
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set used by the column.
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation used by the column.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation used by the column.
COLLATION_NAME	NCHAR VARYING(128)	The name of the collation used by the column.
DOMAIN_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the domain used by the column.
DOMAIN_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the domain used by the column.
DOMAIN_NAME	NCHAR VARYING(128)	The name of the domain used by the column.
USER_DEFINED_TYPE_CATALOG	NCHAR VARYING(128)	The name of the user-defined type catalog.
USER_DEFINED_TYPE_SCHEMA	NCHAR VARYING(128)	The name of the user-defined type schema.
USER_DEFINED_TYPE_NAME	NCHAR VARYING(128)	The name of the user-defined type name.
COLUMN_CARD	BIGINT	Number of unique values in column as set by last UPDATE STATISTICS command for table.

INFORMATION_SCHEMA.CONSTRAINT_COLUMN_USAGE

The CONSTRAINT_COLUMN_USAGE system view lists the table columns on which constraints (referential constraints, unique constraints, check constraints and assertions) that are owned by the current ident are defined.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
COLUMN_NAME	NCHAR VARYING(128)	The name of the table column.

Column name	Data type	Description
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the constraint.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the constraint.

INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE

The CONSTRAINT_TABLE_USAGE system view lists the tables on which constraints (referential constraints, unique constraints, check constraints and assertions) that are owned by the current ident are defined.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the constraint.
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the constraint.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the constraint.

INFORMATION_SCHEMA.DIRECT_SUPERTABLES

The DIRECT_SUPERTABLES system view lists each table created under another table.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
SUPERTABLE_NAME	NCHAR VARYING(128)	The name of the supertable.

INFORMATION_SCHEMA.DIRECT_SUPERTYPES

The DIRECT_SUPERTYPES system view lists each user-defined type created under another user-defined type.

Column name	Data type	Description
UDT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the user-defined type.
UDT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the user-defined type.
UDT_NAME	NCHAR VARYING(128)	The name of the user-defined type.
SUPERTYPE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the supertype.
SUPERTYPE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the supertype.
SUPERTYPE_NAME	NCHAR VARYING(128)	The name of the supertype.

INFORMATION_SCHEMA.DOMAIN_CONSTRAINTS

The DOMAIN_CONSTRAINTS system view lists the domain constraints of domains to which the current ident has access.

Column name	Data type	Description
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the constraint.
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the constraint.
CONSTRAINT_NAME	NCHAR VARYING(128)	Name of the constraint.
DOMAIN_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the domain on which the constraint is defined.
DOMAIN_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the domain on which the constraint is defined.
DOMAIN_NAME	NCHAR VARYING(128)	The name of the domain on which the constraint is defined.
IS_DEFERRABLE	VARCHAR(3)	One of: YES = the constraint is deferrable NO = the constraint is not deferrable.

Column name	Data type	Description
INITIALLY_DEFERRED	VARCHAR(3)	One of: YES = the constraint is immediate NO = the constraint is deferred.

INFORMATION_SCHEMA.DOMAINS

The DOMAINS system view describes each domain to which the current ident has USAGE privilege.

Column name	Data type	Description
DOMAIN_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the domain.
DOMAIN_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the domain.
DOMAIN_NAME	NCHAR VARYING(128)	Name of the domain.
DATA_TYPE	VARCHAR(30)	Identifies the data type of the domain. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP.

Column name	Data type	Description
LOB_MAXIMUM_LENGTH	BIGINT	For the LOB data type, this shows the maximum length in bytes. For all other data types it is the null value.
CHARACTER_MAXIMUM_ LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in characters or bytes. For all other data types it is the null value.
CHARACTER_OCTET_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in octets. For all other data types it is the null value. For single octet character sets, this is the same as CHARACTER_MAX_LENGTH.
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set used by the domain. Null if not CHARACTER data type.
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set used by the domain. Null if not CHARACTER data type.
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set used by the domain. Null if not CHARACTER data type.
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the default collation for the character set. Null if not CHARACTER data type.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the default collation for the character set. Null if not CHARACTER data type.
COLLATION_NAME	NCHAR VARYING(128)	The name of the default collation for the character set. Null if not CHARACTER data type.

Column name	Data type	Description
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of significant digits allowed in the column. For all other data types it is the null value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type (see *Interval Qualifiers* on page *55*). For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision (see *Interval Qualifiers* on page *55*). For other data types it is the null value.

Column name	Data type	Description
DOMAIN_DEFAULT	NCHAR VARYING(200)	This shows the default value for the domain. For more information, see *INFORMATION_SCHEMA.COLUMNS on page 451*

INFORMATION_SCHEMA.EXT_ACCESS_PATHS

The EXT_ACCESS_PATHS view shows all explicit and implicit indexes on tables that are accessible by the current ident. All columns in the indexes are displayed including the primary key columns that are automatically appended to an index.

Column name	Data type	Description
INDEX_CATALOG	NCHAR VARYING(128)	Catalog name for the index.
INDEX_SCHEMA	NCHAR VARYING(128)	Schema name for the index.
INDEX_NAME	NCHAR VARYING(128)	Index name for the secondary index. For implicit indexes this is the name of the constraint that is the reason for the index. This can be constraints such as primary key, unique, foreign key etc.
TABLE_NAME	NCHAR VARYING(128	Name of the table on which the index is defined. The table always has the same catalog and and schema as the index itself.
INDEX_TYPE	VARCHAR(20)	One of: FOREIGN KEY INDEX INTERNAL KEY PRIMARY KEY UNIQUE UNIQUE INDEX.
COLUMN_NAME	NCHAR VARYING(128)	Name of column present in index.
ORDINAL_POSITION	INTEGER	Ordinal position for the column within the index.
COLUMN_SOURCE	VARCHAR(20)	Used to distinguish which columns are the index columns and which columns comes from the primary key of the base table.

Column name	Data type	Description
COLLATION_CATALOG	NCHAR VARYING(128)	Catalog name for the collation used by the index column. The value is null if the column is not of CHARACTER type.
COLLATION_SCHEMA	NCHAR VARYING(128)	Schema name for the collation used by the index column. The value is null if the column is not of CHARACTER type.
COLLATION_NAME	NCHAR VARYING(128)	Name of the collation used by the index column. The value is null if the column is not of CHARACTER type.
INDEX_ALGORITHM	VARCHAR(20)	For secondary index this indicates the type of index. SIMPLE is used for ordinary indexes. Other values include WORD_SEARCH and PINYIN_START. Implicit indexes has null in this column.

INFORMATION_SCHEMA.EXT_COLLATION_DEFINITIONS

The EXT_COLLATION_DEFINITIONS system view shows collation definitions.

Column name	Data type	Description
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation.
COLLATION_NAME	NCHAR VARYING(128)	Name of the collation.
COLLATION_VERSION	VARCHAR(20)	Version of the collation.
COLLATION_DEFINITION	VARCHAR(400)	Delta string defining the collation.
COLLATION_SEQNO	INTEGER	Sequence number.
BASE_COLLATION_CATALOG	NCHAR VARYING(128)	The catalog for the collation on which the current collation is based.

Column name	Data type	Description
BASE_COLLATION_SCHEMA	NCHAR VARYING(128)	The schema for the collation on which the current collation is based.
BASE_COLLATION_NAME	NCHAR VARYING(128)	The name of the collation on which the current collation is based.

INFORMATION_SCHEMA.EXT_COLUMN_OFFSET_INFORMATION

The EXT_COLUMN_OFFSET_INFORMATION system view shows the physical layout of columns in tables accessible by current user.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
TABLE_SYSID	INTEGER	System identifier for the table.
COLUMN_NAME	NCHAR VARYING(128)	Name of the column.
ORDINAL_POSITION	INTEGER	The ordinal position of the column in the table. The first column in the table is number 1.
IS_NULLABLE	CHARACTER VARYING(3)	One of: NO = the column is not nullable, according to the rules in the international standard YES = the null value is allowed in the column.

Column name	Data type	Description
DATA_TYPE	CHARACTER VARYING(30)	Identifies the data type of the domain. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
LOB_MAXIMUM_LENGTH	BIGINT	For the LOB data type, this shows the maximum length in bytes. For all other data types it is the null value.
CHARACTER_MAXIMUM_LENGTH	INTEGER	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.

Column name	Data type	Description
CHARACTER_OCTET_LENGTH	INTEGER	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHARACTER_MAXIMUM_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value.
NUMERIC_SCALE	INTEGER	This defines the total number of significant digits to the right of the decimal point. For INTEGER and SMALLINT, this is 0. For CHARACTER, VARCHAR, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and interval data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_TYPE	CHARACTER VARYING(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.

Column name	Data type	Description
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
COLUMN_OFFSET	INTEGER	Internal offset for column in the table.
COLUMN_LENGTH	INTEGER	Internal length of column.
VARCHAR_OFFSET	INTEGER	Internal offset for field containing actual length for a varying length column.

INFORMATION_SCHEMA.EXT_COLUMN_REMARKS

The EXT_COLUMN_REMARKS system view shows remarks for columns that are accessible by the current ident.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
COLUMN_NAME	NCHAR VARYING(128)	Name of the column.
REMARKS	NCHAR VARYING(254)	Remark for column.

INFORMATION_SCHEMA.EXT_DATABANKS

The EXT_DATABANKS system view shows databanks on which the current ident has table privilege.

An ident with the system privileges BACKUP or SHADOW may see all databanks in the system.

Column name	Data type	Description
DATABANK_CREATOR	NCHAR VARYING(128)	The name of the authorization ident that created the databank.

Column name	Data type	Description
DATABANK_NAME	NCHAR VARYING(128)	The name of the databank.
DATABANK_TYPE	VARCHAR(20)	One of: LOG READ ONLY TEMPORARY TRANSACTION WORK.
IS_ONLINE	VARCHAR(3)	One of: YES = the databank is online NO = the databank is offline.
FILE_NUMBER	INTEGER	Ordinal number for databank file.
FILE_NAME	NCHAR VARYING(256)	Filename for databank.
MAXSIZE	BIGINT	Maximum size for databank file, in kilo bytes
GOALSIZE	BIGINT	Ideal size for databank file, in kilo bytes
MINSIZE	BIGINT	Minimum size for databank file, in kilo bytes
IS_REMOVABLE	VARCHAR(3)	One of: NO = databank is not removable YES = databank is removable
BACKUP_DATE	TIMESTAMP	Last backup date for databank.

INFORMATION_SCHEMA.EXT_IDENTS

The EXT_IDENTS system view shows authorization idents either created by the current ident or those on which the current ident has execute or member privilege.

Column name	Data type	Description
IDENT_CREATOR	NCHAR VARYING(128)	The name of the authorization ident that created the ident.
IDENT_NAME	NCHAR VARYING(128)	The name of the ident.
IDENT_LOGIN	NCHAR VARYING(128)	OS_USER login name. The value is null value if no OS_USER is defined.
HAS_PASSWORD	VARCHAR(3)	This column is only kept for backward compatibility with older versions. The value is always: NO

Column name	Data type	Description
IDENT_TYPE	VARCHAR(20)	One of: GROUP PROGRAM USER.
IDENT_SCHEMA	VARCHAR(3)	One of: YES = the ident has a schema NO = the ident is created without schema

INFORMATION_SCHEMA.EXT_INDEX_COLUMN_USAGE

The EXT_INDEX_COLUMN_USAGE system view shows on which table columns an secondary index is defined. Only indexes defined on table accessible by the current ident is shown.

Column name	Data type	Description
INDEX_CATALOG	NCHAR VARYING(128)	Catalog name for the secondary index.
INDEX_SCHEMA	NCHAR VARYING(128)	Schema name for the secondary index.
INDEX_NAME	NCHAR VARYING(128)	Name of the index.
IS_UNIQUE	VARCHAR(3)	One of: YES = the index may not contain duplicates NO = the index may contain duplicates.
TABLE_NAME	NCHAR VARYING(128)	Name of the table on which the index is defined.
COLUMN_NAME	NCHAR VARYING(128)	Name of column present in index.
IS_ASCENDING	VARCHAR(3)	One of: YES = the sort order of the index is ascending NO = the sort order of the index is descending.
ORDINAL_POSITION	INTEGER	Ordinal position for the column within the index.
COLLATION_CATALOG	NCHAR VARYING(128)	Catalog name for the collation. The value is null if the column is not of CHARACTER type.
COLLATION_SCHEMA	NCHAR VARYING(128)	Schema name for the collation. The value is null if the column is not of CHARACTER type.

Column name	Data type	Description
COLLATION_NAME	NCHAR VARYING(128)	Name of the collation. The value is null if the column is not of CHARACTER type.
INDEX_ALGORITHM	CHARACTER VARYING(20)	Type of index algorithm. SIMPLE or WORD_SEARCH.
INDEX_DATA_TYPE	CHARACTER VARYING(30)	
INDEX_SIZE	INTEGER	
INDEX_SCALE	INTEGER	

INFORMATION_SCHEMA.EXT_INDEXES

The EXT_INDEXES system view shows secondary indexes defined on tables that are accessible by the current ident.

Column name	Data type	Description
INDEX_CATALOG	NCHAR VARYING(128)	Catalog name for the secondary index.
INDEX_SCHEMA	NCHAR VARYING(128)	Schema name for the secondary index.
INDEX_NAME	NCHAR VARYING(128)	Name of the secondary index.
TABLE_NAME	NCHAR VARYING(128)	Name of the table on which the index is defined.
IS_UNIQUE	VARCHAR(3)	One of: YES = the index may not contain duplicates NO = the index may contain duplicates.

INFORMATION_SCHEMA.EXT_OBJECT_IDENT_USAGE

The EXT_OBJECT_IDENT system view shows objects accessible by the current ident.

Column name	Data type	Description
CREATOR_NAME	NCHAR VARYING(128)	Name of authorization ident that created the object.
OBJECT_CATALOG	NCHAR VARYING(128)	Catalog name for the object.
OBJECT_SCHEMA	NCHAR VARYING(128)	Schema name for the object.

Column name	Data type	Description
OBJECT_NAME	NCHAR VARYING(128)	Name of the object.
SPECIFIC_NAME	NCHAR VARYING(128)	Specific name for owned object if object_type is one of FUNCTION METHOD METHOD SPECIFICATION PROCEDURE Otherwise null.
OBJECT_TYPE	VARCHAR(20)	One of: ASSERTION BASE TABLE CHARACTER SET COLLATION CONSTRAINT DATABANK DOMAIN FUNCTION IDENT INDEX METHOD METHOD SPECIFICATION MODULE PROCEDURE SCHEMA SEQUENCE SHADOW STATEMENT SYNONYM TRIGGER USER-DEFINED TYPE VIEW.
CREATION_DATE	TIMESTAMP	Time when object was created.
ALTERATION_DATE	TIMESTAMP	Time when object was last altered.
IS_IMPLICIT	VARCHAR(3)	One of YES - The object has been created implicitly by Mimer when creating other objects NO - The object has been created explicitly by the ident
REMARKS	NCHAR VARYING(254)	Remark for the object.

INFORMATION_SCHEMA.EXT_OBJECT_OBJECT_USED

The EXT_OBJECT_OBJECT_USED system view shows which object that are used by an object. The used and using objects must be accessible to the current ident.

Column name	Data type	Description
USED_OBJECT_TYPE	VARCHAR(20)	Object type for used object. One of: ASSERTION BASE TABLE CHARACTER SET COLLATION DOMAIN FUNCTION INDEX METHOD METHOD SPECIFICATION MODULE PROCEDURE SEQUENCE SHADOW TRIGGER USER-DEFINED TYPE VIEW.
USED_OBJECT_CATALOG	NCHAR VARYING(128)	Catalog name for used object.
USED_OBJECT_SCHEMA	NCHAR VARYING(128)	Schema name for used object.
USED_OBJECT_NAME	NCHAR VARYING(128)	Name of used object
USED_SPECIFIC_NAME	NCHAR VARYING(128)	Specific name for used object

Column name	Data type	Description
USING_OBJECT_TYPE	VARCHAR(20)	Object type for using object. One of: ASSERTION BASE TABLE CHARACTER SET COLLATION CONSTRAINT DOMAIN FUNCTION INDEX METHOD METHOD SPECIFICATION MODULE PROCEDURE SHADOW STATEMENT TRIGGER USER-DEFINED TYPE VIEW.
USING_OBJECT_CATALOG	NCHAR VARYING(128)	Catalog name for using object
USING_OBJECT_SCHEMA	NCHAR VARYING(128)	Schema name for using object
USING_OBJECT_NAME	NCHAR VARYING(128)	Name of using object
USING_SPECIFIC_NAME	NCHAR VARYING(128)	Specific name for using object

INFORMATION_SCHEMA.EXT_OBJECT_OBJECT_USING

The EXT_OBJECT_OBJECT_USING system view shows which object that are using another object. The used and using objects must be accessible to the current ident.

Column name	Data type	Description
USING_OBJECT_TYPE	VARCHAR(20)	Object type for using object. One of: ASSERTION BASE TABLE CHARACTER SET COLLATION CONSTRAINT DOMAIN FUNCTION INDEX METHOD METHOD SPECIFICATION MODULE PROCEDURE SHADOW STATEMENT TRIGGER USER-DEFINED TYPE VIEW.
USING_OBJECT_CATALOG	NCHAR VARYING(128)	Catalog name for using object.
USING_OBJECT_SCHEMA	NCHAR VARYING(128)	Schema name for using object.
USING_OBJECT_NAME	NCHAR VARYING(128)	Name of using object
USING_SPECIFIC_NAME	NCHAR VARYING(128)	Specific name for using object

Column name	Data type	Description
USED_OBJECT_TYPE	VARCHAR(20)	Object type for used object. One of: ASSERTION BASE TABLE CHARACTER SET COLLATION DOMAIN FUNCTION IDENT METHOD METHOD SPECIFICATION MODULE PROCEDURE SEQUENCE SHADOW TRIGGER USER-DEFINED TYPE VIEW.
USED_OBJECT_CATALOG	NCHAR VARYING(128)	Catalog name for used object
USED_OBJECT_SCHEMA	NCHAR VARYING(128)	Schema name for used object
USED_OBJECT_NAME	NCHAR VARYING(128)	Name of used object
USED_SPECIFIC_NAME	NCHAR VARYING(128)	Specific name for used object

INFORMATION_SCHEMA.EXT_OBJECT_PRIVILEGES

The EXT_OBJECT_PRIVILEGES system view shows which object privileges that are granted to an authorization ident. Either GRANTEE or GRANTOR should be equal to current ident.

Column name	Data type	Description
GRANTEE	NCHAR VARYING(128)	Name of authorization ident that has received the privilege.
OBJECT_CATALOG	NCHAR VARYING(128)	Catalog name for object.
OBJECT_SCHEMA	NCHAR VARYING(128)	Schema name for object.
OBJECT_NAME	NCHAR VARYING(128)	Name of object on which privilege is granted.

Column name	Data type	Description
OBJECT_TYPE	VARCHAR(20)	One of: CHARACTER SET DOMAIN DATABANK FUNCTION IDENT METHOD METHOD SPECIFICATION PROCEDURE SEQUENCE STATEMENT USER-DEFINED TYPE.
PRIVILEGE_TYPE	VARCHAR(20)	One of: EXECUTE MEMBER TABLE USAGE USER-DEFINED TYPE.
GRANTOR	NCHAR VARYING(128)	Name of authorization ident that granted the privilege.
IS_GRANTABLE	VARCHAR(3)	One of: YES = the grantee may grant the privilege NO = the grantee may not grant the privilege.

INFORMATION_SCHEMA.EXT_ONEROW

Dummy view containing one single row.

Column name	Data type	Description
M	CHAR(1)	Contains the value M

INFORMATION_SCHEMA.EXT_ROUTINE_MODULE_DEFINITION

The EXT_ROUTINE_MODULE_DEFINITION system view lists definitions for routines that are defined in a module.

Column name	Data type	Description
SPECIFIC_SCHEMA	NCHAR VARYING(128)	The name of the schema to which the routine belongs.
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name of the routine

Column name	Data type	Description
ROUTINE_LENGTH	INTEGER	The total length of the routine definition
ROUTINE_DEFINITION	NCHAR VARYING(400)	The source text for the routine

INFORMATION_SCHEMA.EXT_ROUTINE_MODULE_USAGE

The EXT_ROUTINE_MODULE_USAGE system view lists which routines that are defined in a module.

Column name	Data type	Description
ROUTINE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the routine
ROUTINE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the routine
ROUTINE_NAME	NCHAR VARYING(128)	The name of the routine
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name of the routine
MODULE_NAME	NCHAR VARYING(128)	The name of the module in which the routine is defined

INFORMATION_SCHEMA.EXT_SCHEMAS

The EXT_SCHEMAS system view shows schemas containing objects that are accessible to the current ident.

Column name	Data type	Description
SCHEMA_NAME	NCHAR VARYING(128)	Schema name

INFORMATION_SCHEMA.EXT_SEQUENCES

The EXT_SEQUENCES system view shows sequences that are accessible to the current ident.

Column name	Data type	Description
SEQUENCE_CATALOG	NCHAR VARYING(128)	Catalog name for the sequence.
SEQUENCE_SCHEMA	NCHAR VARYING(128)	Schema name for the sequence.

Column name	Data type	Description
SEQUENCE_NAME	NCHAR VARYING(128)	Name of the sequence.
IS_UNIQUE	VARCHAR(3)	One of: YES = the sequence will yield unique values NO = the sequence may repeat itself.
INITIAL_VALUE	BIGINT	The initial value for the sequence.
INCREMENT	BIGINT	The increment for the sequence.
MAXIMUM_VALUE	BIGINT	The maximum value for the sequence.
FLUSH_RATE	BIGINT	Number of sequence value allocations between saving sequence data to disk.
DATABANK_NAME	NCHAR VARYING(128)	Name of databank where the sequence table is located.

INFORMATION_SCHEMA.EXT_SHADOWS

The EXT_SHADOWS system view shows shadows. A user with shadow privilege may see any shadow or shadows on databanks that are created by the user.

Column name	Data type	Description
SHADOW_CREATOR	NCHAR VARYING(128)	Creator of the shadow.
SHADOW_NAME	NCHAR VARYING(128)	Name of the shadow.
DATABANK_NAME	NCHAR VARYING(128)	Name of databank.
IS_ONLINE	VARCHAR(3)	One of: YES = the shadow is online NO = the shadow is offline.
FILE_NUMBER	INTEGER	Ordinal number for physical file.
FILE_NAME	NCHAR VARYING(256)	Name of physical file for shadow.

INFORMATION_SCHEMA.EXT_SOURCE_DEFINITION

The EXT_SOURCE_DEFINITION system view shows a textual definition for owned objects.

Column name	Data type	Description
OBJECT_CATALOG	NCHAR VARYING(128)	Catalog name for object.

Column name	Data type	Description
OBJECT_SCHEMA	NCHAR VARYING(128)	Schema name for object.
OBJECT_NAME	NCHAR VARYING(128)	Name of object.
SPECIFIC_NAME	NCHAR VARYING(128)	Specific name for routine if object type is one of: CONSTRUCTOR METHOD CONSTRUCTOR FUNCTION FUNCTION INSTANCE METHOD PROCEDURE STATIC METHOD Otherwise the column is null.
COLUMN_NAME	NCHAR VARYING(128)	If object type is BASE TABLE the column contains the name of the column for which the default value is defined. If object type is USER-DEFINED TYPE the column contains the name of the attribute for which the default value is defined. Otherwise the column is null.
OBJECT_TYPE	VARCHAR(20)	One of: CONSTRAINT FUNCTION METHOD METHOD SPECIFICATION MODULE PROCEDURE TRIGGER USER-DEFINED TYPE VIEW.
SOURCE_DEFINITION	NCHAR VARYING(400)	Definition text for object.
SOURCE_LENGTH	INTEGER	Total length of source.
LINE_NUMBER	INTEGER	The line number within the source. 1 = this is the first 400 characters of the source. 2 = this is the second 400 characters of the source, etc.

INFORMATION_SCHEMA.EXT_STATEMENTS

The EXT_STATEMENTS view shows all precompiled statements available to the current IDENT.

Column name	Data type	Description
STATEMENT_CATALOG	NCHAR VARYING(128)	Catalog name for precompiled statement.
STATEMENT_SCHEMA	NCHAR VARYING(128)	Schema name for precompiled statement.
STATEMENT_NAME	NCHAR VARYING(128)	Name of precompiled statement.
STATEMENT_TYPE	INTEGER	Type of precompiled statement.
STATEMENT_DEFINITION	NCHAR VARYING(200)	Definition text of precompiled statement. If the definition does not fit, the null value is shown. In that case the definition can be found in INFORMATION_SCHEMA.EXT_STATEMENT_DEFINITION.
IS_SCROLLABLE	VARCHAR(3)	One of: YES = a scrollable version of the statement exists. NO = no scrollable version of the statement exists.
IS_FORWARD_ONLY	VARCHAR(3)	One of: YES = a forward only version of the statement exists. NO = no forward only version of the statement exists.

INFORMATION_SCHEMA.EXT_STATEMENT_DEFINITION

The EXT_STATEMENT_DEFINITION view shows a textual definition of the precompiled statements available to the current IDENT.

Column name	Data type	Description
STATEMENT_SCHEMA	NCHAR VARYING(128)	Schema name for precompiled statement.
STATEMENT_NAME	NCHAR VARYING(128)	Name of precompiled statement.
STATEMENT_SEQUENCE_NO	INTEGER	The sequence number.

Column name	Data type	Description
STATEMENT_DEFINITION	NCHAR VARYING(400)	The definition text of the precompiled statement. 400 characters for each sequence number.

INFORMATION_SCHEMA.EXT_STATISTICS

The EXT_STATISTICS system view shows when statistics for a base table was collected.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	Catalog name for base table.
TABLE_SCHEMA	NCHAR VARYING(128)	Schema name for base table.
TABLE_NAME	NCHAR VARYING(128)	Name of base table.
STATISTICS_GATHERED	TIMESTAMP(2)	Time when statistics for this table was collected
CARDINALITY	BIGINT	Number of rows in table when statistics was gathered.

INFORMATION_SCHEMA.EXT_SYNONYMS

The EXT_SYNONYMS system view shows synonyms on accessible tables.

Column name	Data type	Description
SYNONYM_CATALOG	NCHAR VARYING(128)	Catalog name for synonym.
SYNONYM_SCHEMA	NCHAR VARYING(128)	Schema name for synonym.
SYNONYM_NAME	NCHAR VARYING(128)	Name of synonym.
TABLE_CATALOG	NCHAR VARYING(128)	Catalog for table on which synonym is defined.
TABLE_SCHEMA	NCHAR VARYING(128)	Schema name for table.
TABLE_NAME	NCHAR VARYING(128)	Name of table.

INFORMATION_SCHEMA.EXT_SYSTEM_PRIVILEGES

The EXT_SYSTEM_PRIVILEGES system view shows granted to or by the current ident.

Column name	Data type	Description
GRANTEE	NCHAR VARYING(128)	Name of grantee.
PRIVILEGE_TYPE	VARCHAR(20)	One of: BACKUP DATABANK IDENT SCHEMA SHADOW STATISTICS.
GRANTOR	NCHAR VARYING(128)	Name of grantor.
IS_GRANTABLE	VARCHAR(3)	One of: YES = grantee has grant option NO = grantee has not grant option.

INFORMATION_SCHEMA.EXT_TABLE_DATABANK_USAGE

The EXT_TABLE_DATABANK_USAGE system view shows in which databank a base table is located. Base tables accessible to the current ident are shown.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	Catalog for table.
TABLE_SCHEMA	NCHAR VARYING(128)	Schema name for table.
TABLE_NAME	NCHAR VARYING(128)	Name of table.
DATABANK_CREATOR	NCHAR VARYING(128)	Name of authorization ident that created databank.
DATABANK_NAME	NCHAR VARYING(128)	Name of databank.

INFORMATION_SCHEMA.KEY_COLUMN_USAGE

The KEY_COLUMN_USAGE system view lists the table columns that are constrained as keys and on accessible tables.

Column name	Data type	Description
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table constraint.
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the table constraint.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table or view.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view.
TABLE_NAME	NCHAR VARYING(128)	The name of the table or view.
COLUMN_NAME	NCHAR VARYING(128)	The name of the column.
ORDINAL_POSITION	INTEGER	The ordinal position of the column within the key.

INFORMATION_SCHEMA.METHOD_SPECIFICATION_PARAMETERS

Contains one row for each parameter for a method specification belonging to a user-defined type on which the current user has USAGE privilege.

Column name	Data type	Description
SPECIFIC_CATALOG	NCHAR VARYING(128)	Name of catalog containing method specification.
SPECIFIC_SCHEMA	NCHAR VARYING(128)	Name of schema containing method specification.
SPECIFIC_NAME	NCHAR VARYING(128)	Name of method specification
ORDINAL_POSITION	INTEGER	Ordinal position for parameter
PARAMETER_MODE	VARCHAR(5)	One of IN = The parameter mode is in.

Column name	Data type	Description
IS_RESULT	VARCHAR(3)	One of YES = The parameter is a result parameter NO = The parameter is not a result parameter
PARAMETER_NAME	NCHAR VARYING(128)	Name of parameter
DATA_TYPE	VARCHAR(30)	The type of the parameter. One of: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHARACTER LARGE OBJECT NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED
CHARACETR_MAXIMUM_LENGTH	INTEGER	For a character data type, this shows the maximum length in characters. For all other data types it is the null value.
CHARACTER_OCTET_LENGTH	INTEGER	For a character data type, this shows the maximum length in octets. For all other data types it is the null value. It the same value as CHARACTER_MAXIMUM length for single octet data types.
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set used by the parameter
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set used by the parameter

Column name	Data type	Description
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set used by the parameter
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation used by the parameter.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation used by the parameter.
COLLATION_NAME	NCHAR VARYING(128)	The name of the collation used by the parameter.
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of significant digits allowed in the column. For all other data types it is the null value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.

Column name	Data type	Description
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier. Can be one of: YEAR YEAR TO MONTH DAY HOUR MINUTE SECOND DAY TO HOUR DAY TO MINUTE DAY TO SECOND HOUR TO MINUTE HOUR TO SECOND MINUTE TO SECOND. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision (see Interval Qualifiers). For other data types it is the null value.
USER_DEFINED_TYPE_CATALOG	NCHAR VARYING(128)	The name of the user-defined type catalog used by the parameter.
USER_DEFINED_TYPE_SCHEMA	NCHAR VARYING(128)	The name of the user-defined type schema used by the parameter.
USER_DEFINED_TYPE_NAME	NCHAR VARYING(128)	The name of the user-defined type name used by the parameter
PARAMETER_DOMAIN_CATALOG	NCHAR VARYING(128)	The catalog name for a domain when domain is used as the type for a parameter in a method.
PARAMETER_DOMAIN_SCHEMA	NCHAR VARYING(128)	The schema name for a domain when domain is used as the type for a parameter in a method.
PARAMETER_DOMAIN_NAME	NCHAR VARYING(128)	The name for a domain when domain is used as the type for a parameter in a method.

INFORMATION_SCHEMA.METHOD_SPECIFICATIONS

Contains one row for each method specification belonging to a user-defined type on which the current user has USAGE privilege.

Column name	Data type	Description
SPECIFIC_CATALOG	NCHAR VARYING(128)	Catalog name for method specification
SPECIFIC_SCHEMA	NCHAR VARYING(128)	Schema name for method specification
SPECIFIC_NAME	NCHAR VARYING(128)	Specific name for method specification
UDT_CATALOG	NCHAR VARYING(128)	Catalog for user-defined type containing method specification
UDT_SCHEMA	NCHAR VARYING(128)	Schema for user-defined type containing method specification
UDT_NAME	NCHAR VARYING(128)	Name of user-defined type containing method specification
METHOD_NAME	NCHAR VARYING(128)	Method name
IS_STATIC	VARCHAR(3)	One of YES = Method specification is static NO = Method specification is not static
IS_OVERRIDING	VARCHAR(3)	One of YES = Method specification is overriding NO = Method specification is not overriding
IS_CONSTRUCTOR	VARCHAR(3)	One of YES = Method specification is a constructor method NO = Method specification is not a constructor method

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	The return data type of the method specification. One of: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHARACTER LARGE OBJECT NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED
LOB_MAXIMUM_ LENGTH	INTEGER	For a lob data type, this shows the maximum length. For all other data types it is the null value.
CHARACTER_MAXIMUM_ LENGTH	INTEGER	For a character data type, this shows the maximum length in characters. For all other data types it is the null value.
CHARACTER_OCTET_ LENGTH	INTEGER	For a character data type, this shows the maximum length in octets. For all other data types it is the null value. It the same value as CHARACTER_MAXIMUM length for single octet data types.
CHARACTER_SET_ CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set used by the parameter
CHARACTER_SET_ SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set used by the result type for the method specification

Column name	Data type	Description
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set used by the result type for the method specification
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation used by the result type for the method specification.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation used by the result type for the method specification.
COLLATION_NAME	NCHAR VARYING(128)	The name of the collation used by the result type for the method specification.
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of significant digits allowed in the column. For all other data types it is the null value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.

Column name	Data type	Description
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier. Can be one of: YEAR YEAR TO MONTH DAY HOUR MINUTE SECOND DAY TO HOUR DAY TO MINUTE DAY TO SECOND HOUR TO MINUTE HOUR TO SECOND MINUTE TO SECOND. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision (see Interval Qualifiers). For other data types it is the null value.
RETURN_UDT_CATALOG	NCHAR VARYING(128)	The name of the user-defined type catalog used by the result type for the method specification.
RETURN_UDT_SCHEMA	NCHAR VARYING(128)	The name of the user-defined type schema used by the result type for the method specification.
RETURN_UDT_NAME	NCHAR VARYING(128)	The name of the user-defined type name used by the result type for the method specification
METHOD_LANGUAGE	VARCHAR(20)	One of: SQL
IS_DETERMINISTIC	VARCHAR(3)	One of: YES = the function was declared as DETERMINISTIC when it was created NO = the function was not declared as DETERMINISTIC when it was created.
SQL_DATA_ACCESS	VARCHAR(20)	One of: CONTAINS SQL READS SQL MODIFIES SQL.

Column name	Data type	Description
IS_NULL_CALL	VARCHAR(3)	One of: YES = The method will be invoked even if any parameter is null NO = The method will return NULL if any parameter is null
CREATED	TIMESTAMP(2)	The date and time at which the method specification was created
LAST_ALTERED	TIMESTAMP(2)	The date and time at which the method specification was last altered. If the method has not been altered, this value will be null.
AS_LOCATOR	VARCHAR(3)	One of: YES = The returned data type for the method is declared as a locator NO = The returned data type for the method is not declared as a locator
RETURN_DOMAIN_CATALOG	NCHAR VARYING(128)	The catalog name for a domain when domain is used as return type for a method.
RETURN_DOMAIN_SCHEMA	NCHAR VARYING(128)	The schema name for a domain when domain is used as return type for a method.
RETURN_DOMAIN_NAME	NCHAR VARYING(128)	The name for a domain when domain is used as return type for a method.

INFORMATION_SCHEMA.MODULES

The MODULES system views shows modules created by the current ident.

Column name	Data type	Description
MODULE_CATALOG	NCHAR VARYING(128)	Catalog name for module.
MODULE_SCHEMA	NCHAR VARYING(128)	Schema name for module.
MODULE_NAME	NCHAR VARYING(128)	Name of module.
DEFAULT_CHARACTER_SET_CATALOG	NCHAR VARYING(128)	Catalog name for default character set.
DEFAULT_CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	Schema name for default character set.
DEFAULT_CHARACTER_SET_NAME	NCHAR VARYING(128)	The name for default character set.

Column name	Data type	Description
DEFAULT_SCHEMA_CATALOG	NCHAR VARYING(128)	Catalog name for default schema of the module.
DEFAULT_SCHEMA_NAME	NCHAR VARYING(128)	Name of default schema of the module.
MODULE_DEFINITION	NCHAR VARYING(400)	Module definition; if text larger than 400, the null value is stored. The complete text can always be found in the EXT_SOURCE_ DEFINITION view.
SQL_PATH	NCHAR VARYING(200)	The default path for the module

INFORMATION_SCHEMA.PARAMETERS

The PARAMETERS system view lists the parameters of routines on which the current ident has EXECUTE privilege.

Column name	Data type	Description
SPECIFIC_CATALOG	NCHAR VARYING(128)	The catalog name for the specific name of the routine.
SPECIFIC_SCHEMA	NCHAR VARYING(128)	The schema name for the specific name of the routine.
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name of the routine.
ORDINAL_POSITION	INTEGER	The ordinal position of the parameter in the routine. The first parameter in the routine is number 1.
PARAMETER_MODE	VARCHAR(5)	Indicates whether the parameter is IN, OUT or INOUT.
PARAMETER_NAME	NCHAR VARYING(128)	The name of the parameter.

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	The data type of the parameter. One of: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHARACTER LARGE OBJECT NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED
LOB_MAXIMUM_LENGTH	BIGINT	For the LOB data type, this shows the maximum length in bytes. For all other data types it is the null value.
CHARACTER_MAXIMUM_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in characters or bytes. For all other data types it is the null value.
CHARACTER_OCTET_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHARACTER_MAX_LENGTH).
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set.
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set.

Column name	Data type	Description
CHARACTER_SET_NAME	NCHAR VARYING(128)	Name of the character set.
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation.
COLLATION_NAME	NCHAR VARYING(128)	Name of the collation.
USER_DEFINED_TYPE_CATALOG	NCHAR VARYING(128)	The name of the user-defined type catalog.
USER_DEFINED_TYPE_SCHEMA	NCHAR VARYING(128)	The name of the user-defined type schema.
USER_DEFINED_TYPE_NAME	NCHAR VARYING(128)	The name of the user-defined type name.
NUMERIC_PRECISION	INTEGER	For numeric data types, this shows the total number of significant digits allowed in the column. For all other data types it is the null value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and interval data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.

Column name	Data type	Description
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type (see *Interval Qualifiers* on page *55*).
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision (*Interval Qualifiers* on page *55*).
PARAMETER_DEFAULT	NCHAR VARYING(200)	Default value for parameter.
DOMAIN_CATALOG	NCHAR VARYING(128)	The catalog name for a domain when domain is used as the type for a parameter in a routine.
DOMAIN_SCHEMA	NCHAR VARYING(128)	The schema name for a domain when domain is used as the type for a parameter in a routine.
DOMAIN_NAME	NCHAR VARYING(128)	The name for a domain when domain is used as the type for a parameter in a routine.

INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS

The REFERENTIAL_CONSTRAINTS system view lists the referential constraints that are accessible by the current ident.

Column name	Data type	Description
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the referential constraint.
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the referential constraint.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the referential constraint.
UNIQUE_CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the unique constraint being referenced.
UNIQUE_CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the unique constraint being referenced
UNIQUE_CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the unique constraint being referenced.

Column name	Data type	Description
MATCH_OPTION	VARCHAR(20)	One of: NONE PARTIAL FULL.
UPDATE_RULE	VARCHAR(20)	One of: CASCADE SET NULL SET DEFAULT NO ACTION.
DELETE_RULE	VARCHAR(20)	One of: CASCADE SET NULL SET DEFAULT NO ACTION.

INFORMATION_SCHEMA.ROUTINE_COLUMN_USAGE

The ROUTINE_COLUMN_USAGE system view lists the table columns that are owned by the current ident which are referenced from within a routine.

Column name	Data type	Description
SPECIFIC_CATALOG	NCHAR VARYING(128)	The catalog name for the specific name of the routine.
SPECIFIC_SCHEMA	NCHAR VARYING(128)	The schema name for the specific name of the routine.
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name of the routine.
ROUTINE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the routine.
ROUTINE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the routine.
ROUTINE_NAME	NCHAR VARYING(128)	The name of the routine.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
COLUMN_NAME	NCHAR VARYING(128)	The name of the column.

INFORMATION_SCHEMA.ROUTINE_PRIVILEGES

The ROUTINE_PRIVILEGES system view lists privileges that were granted by the current ident, and privileges that were granted to the current ident or to PUBLIC, on a routine.

Column name	Data type	Description
GRANTOR	NCHAR VARYING(128)	The name of the ident who granted the privilege.
GRANTEE	NCHAR VARYING(128)	The name of the ident to whom the privilege was granted. Granting a privilege to PUBLIC will result in only one row (per privilege granted) in this view and the name PUBLIC will be shown.
SPECIFIC_CATALOG	NCHAR VARYING(128)	The catalog name for the specific name of the routine.
SPECIFIC_SCHEMA	NCHAR VARYING(128)	The schema name for the specific name of the routine.
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name of the routine.
ROUTINE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the routine.
ROUTINE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the routine.
ROUTINE_NAME	NCHAR VARYING(128)	The name of the routine.
PRIVILEGE_TYPE	VARCHAR(20)	The type of the privilege.
IS_GRANTABLE	VARCHAR(3)	One of: YES = the privilege is held with the WITH GRANT OPTION NO = the privilege is held without the WITH GRANT OPTION.

INFORMATION_SCHEMA.ROUTINE_TABLE_USAGE

The ROUTINE_TABLE_USAGE system view lists the tables that are owned by the current ident on which SQL-invoked routines depend.

Column name	Data type	Description
SPECIFIC_CATALOG	NCHAR VARYING(128)	The catalog name for the specific name of the routine.
SPECIFIC_SCHEMA	NCHAR VARYING(128)	The schema name for the specific name of the routine.

Column name	Data type	Description
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name of the routine.
ROUTINE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the routine.
ROUTINE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the routine.
ROUTINE_NAME	NCHAR VARYING(128)	The name of the routine.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.

INFORMATION_SCHEMA.ROUTINES

The ROUTINES system view lists the routines on which the current ident has EXECUTE privilege. If the routine is a function, the result data type for the function is returned.

Column name	Data type	Description
SPECIFIC_CATALOG	NCHAR VARYING(128)	The catalog name for the specific name of the routine.
SPECIFIC_SCHEMA	NCHAR VARYING(128)	The schema name for the specific name of the routine.
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name of the routine.
ROUTINE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the routine.
ROUTINE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the routine.
ROUTINE_NAME	NCHAR VARYING(128)	The name of the routine.
MODULE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the module to which the routine belongs.
MODULE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the module to which the routine belongs.
MODULE_NAME	NCHAR VARYING(128)	The name of the module to which the routine belongs.

Column name	Data type	Description
ROUTINE_TYPE	VARCHAR(20)	One of: CONSTRUCTOR METHOD FUNCTION INSTANCE METHOD PROCEDURE STATIC METHOD
DATA_TYPE	VARCHAR(30)	The data type returned by the function or method. One of: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHARACTER LARGE OBJECT NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED
LOB_MAXIMUM_LENGTH	BIGINT	For LOB data types, this shows the maximum length in bytes.
CHARACTER_MAXIMUM_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in characters or bytes. For all other data types it is the null value.

Column name	Data type	Description
CHARACTER_OCTET_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHARACTER_MAX_LENGTH).
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set.
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set.
CHARACTER_SET_NAME	NCHAR VARYING(128)	Name of the character set.
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation.
COLLATION_NAME	NCHAR VARYING(128)	Name of the collation.
USER_DEFINED_TYPE_CATALOG	NCHAR VARYING(128)	The name of the user-defined type catalog.
USER_DEFINED_TYPE_SCHEMA	NCHAR VARYING(128)	The name of the user-defined type schema.
USER_DEFINED_TYPE_NAME	NCHAR VARYING(128)	The name of the user-defined type name.
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of significant digits contained in the result. For all other data types it is the null value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.

Column name	Data type	Description
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and interval data types, this result contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_TYPE	VARCHAR(30)	For interval data types, this is a character string specifying the interval qualifier for the named interval data type (see *Interval Qualifiers* on page *55*).
INTERVAL_PRECISION	INTEGER	For interval data types, this is the number of significant digits for the interval leading precision, see *Interval Qualifiers* on page *55*.
EXTERNAL_LANGUAGE	VARCHAR(20)	The language for the routine if it is an external routine, otherwise the null value is shown
IS_DETERMINISTIC	VARCHAR(3)	One of: YES = the function was declared as DETERMINISTIC when it was created NO = the function was not declared as DETERMINISTIC when it was created.
SQL_DATA_ACCESS	VARCHAR(20)	One of: CONTAINS SQL READS SQL MODIFIES SQL.
IS_NULL_CALL	VARCHAR(3)	If the routine is a function or method then one of: YES = The function will be invoked even if any parameter is null NO = The function will return NULL if any parameter is null

Column name	Data type	Description
ROUTINE_BODY	VARCHAR(20)	One of: SQL = the routine is an SQL routine EXTERNAL = the routine is an external routine.
ROUTINE_DEFINITION	NCHAR VARYING(200)	The text of the routine definition. If the actual definition would not fit into the maximum length of this column, the null value will be shown and the definition text will appear in the view INFORMATION_SCHEMA.EXT_SOURCE_DEFINITION.
EXTERNAL_NAME	NCHAR VARYING(128)	The external name of the routine if it is an external routine, otherwise the null value is shown.
PARAMETER_STYLE	VARCHAR(20)	The parameter passing style of the routine if it is an external routine, otherwise the null value is shown.
SQL_PATH	NCHAR VARYING(200)	The path for the routine. The null value is shown if no path is defined.
SCHEMA_LEVEL_ROUTINE	VARCHAR(3)	One of: YES = the routine was created on its own NO = the routine was created in a module.
IS_RESULT	VARCHAR(3)	One of: YES = the routine returns a result set NO = the routine does not return a result set.
EXTERNAL_LIBRARY	NCHAR VARYING(128)	
DOMAIN_CATALOG	NCHAR VARYING(128)	The catalog name for a domain when domain is used as return type for a function or a result set procedure.
DOMAIN_SCHEMA	NCHAR VARYING(128)	The schema name for a domain when domain is used as return type for a function or a result set procedure.

Column name	Data type	Description
DOMAIN_NAME	NCHAR VARYING(128)	The name for a domain when domain is used as return type for a function or a result set procedure.

INFORMATION_SCHEMA.SCHEMATA

The SCHEMATA system view lists the schemas that are owned by the current ident.

Column name	Data type	Description
CATALOG_NAME	NCHAR VARYING(128)	The name of the catalog containing the schema.
SCHEMA_NAME	NCHAR VARYING(128)	The name of the schema.
SCHEMA_OWNER	NCHAR VARYING(128)	The name of the ident who created the schema.
DEFAULT_CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog that contains the default character set for the schema.
DEFAULT_CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema that contains the default character set for the schema.
DEFAULT_CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the default character set for the schema.
SQL_PATH	NCHAR VARYING(200)	The default path for the schema

INFORMATION_SCHEMA.SEQUENCES

The SEQUENCES system view shows sequences that are accessible to the current ident.

Column name	Data type	Description
SEQUENCE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the sequence.
SEQUENCE_SCHEMA	NCHAR VARYING(128)	Schema name for the sequence.
SEQUENCE_NAME	NCHAR VARYING(128)	Name of the sequence.
DATA_TYPE	VARCHAR(30)	Data type for sequence. One of: SMALLINT INTEGER BIGINT

Column name	Data type	Description
NUMERIC_PRECISION	INTEGER	Number of significant digits for sequence value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown.
NUMERIC_SCALE	INTEGER	Number of significant decimals. This value will always be zero.
START_VALUE	VARCHAR(20)	Initial value for sequence.
MINIMUM_VALUE	VARCHAR(20)	Minimum value for sequence.
MAXIMUM_VALUE	VARCHAR(20)	Maximum value for sequence.
INCREMENT	VARCHAR(20)	Increment value for sequence.
CYCLE_OPTION	VARCHAR(3)	Indicates whether sequence is defined with cycle option or not. One of: YES NO

INFORMATION_SCHEMA.SQL_FEATURES

The SQL_FEATURES lists the features and subfeatures of the SQL-2011 standard, and indicates which of these Mimer SQL supports.

Column name	Data type	Description
FEATURE_ID	VARCHAR(20)	Identifies the feature by a letter and three digits.
FEATURE_NAME	VARCHAR(254)	Short description of the feature.
SUB_FEATURE_ID	VARCHAR(20)	Identifies a subfeature by two digits. Single space if feature described.
SUB_FEATURE_NAME	VARCHAR(254)	Short description of the subfeature. Single space if feature described.
IS_SUPPORTED	VARCHAR(3)	YES if fully supported by Mimer SQL, otherwise NO.
IS_VERIFIED_BY	VARCHAR(3)	Should identify the test used to verify the conformance (always null).
IS_CORE_SQL	VARCHAR(3)	YES if the feature belongs to Core SQL, otherwise NO.

Column name	Data type	Description
COMMENTS	VARCHAR(254)	Comments pertinent to the feature element.

INFORMATION_SCHEMA.SQL_LANGUAGES

The SQL_LANGUAGES system view lists the conformance levels, options and dialects supported by the SQL implementation.

Column name	Data type	Description
SQL_LANGUAGE_SOURCE	VARCHAR(254)	The organization that defined the SQL version.
SQL_LANGUAGE_YEAR	VARCHAR(254)	The year the relevant source document was approved.
SQL_LANGUAGE _CONFORMANCE	VARCHAR(254)	The conformance level to the relevant document that the implementation claims.
SQL_LANGUAGE _INTEGRITY	VARCHAR(254)	(Meaning no longer defined).
SQL_LANGUAGE _IMPLEMENTATION	VARCHAR(254)	A character string, defined by the vendor, that uniquely defines the vendor's SQL product.
SQL_LANGUAGE _BINDING_STYLE	VARCHAR(254)	Included to envisage future adoption of direct, module or other binding styles.
SQL_LANGUAGE _PROGRAMMING _LANGUAGE	VARCHAR(254)	The host language for which the binding style is supported.

INFORMATION_SCHEMA.SQL_SIZING

The SQL_SIZING view lists the sizing items defined in the SQL-2011 standard and, for each of these, indicate the size supported by Mimer SQL.

Column name	Data type	Description
SIZING_ID	SMALLINT	Identifies the sizing item by an integer value.
SIZING_NAME	VARCHAR(50)	Description of the sizing item.
SUPPORTED_VALUE	INTEGER	0 if no limit by Mimer SQL null value if item not supported by Mimer SQL
COMMENTS	VARCHAR(200)	Comments pertinent to the sizing item.

INFORMATION_SCHEMA.TABLE_CONSTRAINTS

The TABLE_CONSTRAINTS system view lists the table constraints that are accessible by the current ident.

Column name	Data type	Description
CONSTRAINT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table constraint.
CONSTRAINT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view.
CONSTRAINT_NAME	NCHAR VARYING(128)	The name of the table constraint.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table or view.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view.
TABLE_NAME	NCHAR VARYING(128)	The name of the table or view.
CONSTRAINT_TYPE	VARCHAR(20)	One of: CHECK FOREIGN KEY PRIMARY KEY UNIQUE.
IS_DEFERRABLE	VARCHAR(3)	One of: YES = the constraint is deferrable NO = the constraint is not deferrable.
INITIALLY_DEFERRED	VARCHAR(3)	One of: YES = the constraint is immediate NO = the constraint is deferred.

INFORMATION_SCHEMA.TABLE_PRIVILEGES

The TABLE_PRIVILEGES system view lists privileges that were granted by the current ident, and privileges that were granted to the current ident or to PUBLIC, on an entire table.

Column name	Data type	Description
GRANTOR	NCHAR VARYING(128)	The name of the ident who granted the privilege.

Column name	Data type	Description
GRANTEE	NCHAR VARYING(128)	The name of the ident to whom the privilege was granted. Granting a privilege to PUBLIC will result in only one row (per privilege granted) in this view and the name PUBLIC will be shown.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table in question.
PRIVILEGE_TYPE	VARCHAR(20)	A value describing the type of the privilege. One of: DELETE INSERT REFERENCES SELECT UPDATE. Rows where the privilege type is REFERENCES, UPDATE or INSERT only describe cases where the grantee was granted the privilege on the entire table. Where the GRANT statement granted REFERENCES or UPDATE privilege to specified columns of a table, no rows appear in TABLE_PRIVILEGES but there are rows in COLUMN_PRIVILEGES. Note that when multiple table privileges are granted to the same user at the same time (e.g. when the keyword ALL is used), multiple rows appear in this view (one for each privilege granted).
IS_GRANTABLE	VARCHAR(3)	One of: YES = the privilege is held with the WITH GRANT OPTION NO = the privilege is held without the WITH GRANT OPTION.

INFORMATION_SCHEMA.TABLES

The TABLES system view lists tables to which the current ident has access.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table or view.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view.
TABLE_NAME	NCHAR VARYING(128)	The name of the table or view.
TABLE_TYPE	VARCHAR(20)	One of: BASE TABLE = the row describes a table VIEW = the row describes a view.
COMMIT_ACTION	VARCHAR(20)	Indicates what happens with records in a temporary table at commit. One of: DELETE PRESERVE The column will be null if the table is not temporary.

INFORMATION_SCHEMA.TRANSLATIONS

The TRANSLATIONS system view lists the character translations on which the current ident has USAGE privilege.

The source character set is the character set to which the characters that are to be translated by the translation belong.

The target character set is the character set to which the characters that are the result of the translation belong.

Column name	Data type	Description
TRANSLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the translation.
TRANSLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the translation.
TRANSLATION_NAME	NCHAR VARYING(128)	The name of the translation.
SOURCE_CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the source character set.
SOURCE_CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the source character set.
SOURCE_CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the source character set.

Column name	Data type	Description
TARGET_CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the target character set.
TARGET_CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the target character set.
TARGET_CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the target character set.

INFORMATION_SCHEMA.TRIGGERED_UPDATE_COLUMNS

The TRIGGERED_UPDATE_COLUMNS system view lists the columns owned by the current ident that are referenced from the explicit column list in the trigger event of an UPDATE trigger.

Column name	Data type	Description
TRIGGER_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the trigger.
TRIGGER_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the trigger.
TRIGGER_NAME	NCHAR VARYING(128)	The name of the trigger.
EVENT_OBJECT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table or view on which the trigger is created.
EVENT_OBJECT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view on which the UPDATE trigger is created.
EVENT_OBJECT_TABLE	NCHAR VARYING(128)	The name of the table or view on which the UPDATE trigger is created.
EVENT_OBJECT_COLUMN	NCHAR VARYING(128)	The name of the table or view column referenced in the column list of the UPDATE trigger event.

INFORMATION_SCHEMA.TRIGGER_COLUMN_USAGE

The TRIGGER_COLUMN_USAGE view lists the table columns on which triggers, that owned by the current ident, depend because they are referenced in the search condition of the trigger or in one of the statements in the body of the trigger.

Column name	Data type	Description
TRIGGER_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the trigger.
TRIGGER_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the trigger.
TRIGGER_NAME	NCHAR VARYING(128)	The name of the trigger.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the referenced table or view.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the referenced table or view.
TABLE_NAME	NCHAR VARYING(128)	The name of the referenced table or view.
COLUMN_NAME	NCHAR VARYING(128)	The name of the referenced column.

INFORMATION_SCHEMA.TRIGGER_TABLE_USAGE

The TRIGGER_TABLE_USAGE view lists the tables on which triggers, that owned by the current ident, depend.

Column name	Data type	Description
TRIGGER_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the trigger.
TRIGGER_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the trigger.
TRIGGER_NAME	NCHAR VARYING(128)	The name of the trigger.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the referenced table or view.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the referenced table or view.
TABLE_NAME	NCHAR VARYING(128)	The name of the referenced table or view.

INFORMATION_SCHEMA.TRIGGERS

The TRIGGERS system view lists the triggers owned by the current ident.

Column name	Data type	Description
TRIGGER_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the trigger.
TRIGGER_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the trigger.
TRIGGER_NAME	NCHAR VARYING(128)	The name of the trigger.
EVENT_MANIPULATION	VARCHAR(20)	The data manipulation event triggering execution of the trigger (the trigger event). One of: INSERT DELETE UPDATE.
EVENT_OBJECT_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table or view on which the trigger is created.
EVENT_OBJECT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view on which the trigger is created.
EVENT_OBJECT_TABLE	NCHAR VARYING(128)	The name of the table or view on which the trigger is created.
ACTION_ORDER	INTEGER	Ordinal number for trigger execution. This number will define the execution order of triggers on the same table and with the same value for EVENT_MANIPULATION, ACTION_CONDITION, CONDITION_TIMING and ACTION_ORIENTATION. The trigger with 1 in this column will be executed first, followed by the trigger with 2, etc.
ACTION_CONDITION	NCHAR VARYING(200)	The character representation of the search condition in the WHEN clause of the trigger. If the length of the text exceeds 200 characters, the null value will be shown.
ACTION_STATEMENT	NCHAR VARYING(200)	The character representation of the body of the trigger. If the length of the text exceeds 200 characters, the null value will be shown.

Column name	Data type	Description
ACTION_ORIENTATION	VARCHAR(20)	One of: ROW = the trigger is a row trigger STATEMENT = the trigger is a statement trigger.
ACTION_TIMING	VARCHAR(20)	One of: BEFORE = the trigger is executed before the triggering data manipulation operation INSTEAD OF = the trigger is executed instead of the triggering data manipulation operation AFTER = the trigger is executed after the triggering data manipulation operation.
ACTION_REFERENCE_OLD_TABLE	NCHAR VARYING(128)	The identifier specified in the OLD TABLE clause.
ACTION_REFERENCE_NEW_TABLE	NCHAR VARYING(128)	The identifier specified in the NEW TABLE clause.
ACTION_REFERENCE_OLD_ROW	NCHAR VARYING(128)	The identifier specified in the OLD ROW clause.
ACTION_REFERENCE_NEW_ROW	NCHAR VARYING(128)	The identifier specified in the NEW ROW clause.
COLUMN_LIST_IS_IMPLICIT	CHAR(3)	One of: YES = the trigger will be executed on update of any column in the table NO = the trigger will only be executed on update of those columns specified in the UPDATE OF clause in the trigger definition.
CREATED	TIMESTAMP(2)	The date when the trigger was created.

INFORMATION_SCHEMA.UDT_PRIVILEGES

Contains one row for each user-defined type on which the current user has granted or been granted USAGE privilege.

Column name	Data type	Description
GRANTOR	NCHAR VARYING(128)	Name of authorization ident who has granted privilege
GRANTEE	NCHAR VARYING(128)	Name of authorization ident who has been granted privilege

Column name	Data type	Description
UDT_CATALOG	NCHAR VARYING(128)	Catalog name for user-defined type.
UDT_SCHEMA	NCHAR VARYING(128)	Schema name for user-defined type.
UDT_NAME	NCHAR VARYING(128)	Name of user-defined type
PRIVILEGE_TYPE	VARCHAR(20)	One of USAGE TYPE
IS_GRANTABLE	VARCHAR(3)	One of YES = The grantee has the privilege with grant option. NO = The grantee has the privilege without grant option.

INFORMATION_SCHEMA.USAGE_PRIVILEGES

The USAGE_PRIVILEGES system view lists the USAGE privileges that were granted by the current ident, and granted to the current ident or to PUBLIC.

Column name	Data type	Description
GRANTOR	NCHAR VARYING(128)	The name of the ident who granted the privilege.
GRANTEE	NCHAR VARYING(128)	The name of the ident to whom the privilege was granted. Granting a privilege to PUBLIC will result in only one row (per privilege granted) in this view and the name PUBLIC will be shown.
OBJECT_CATALOG	NCHAR VARYING(128)	The name of the catalog that contains the object character set or collation.
OBJECT_SCHEMA	NCHAR VARYING(128)	The name of the schema that contains the object character set or collation.
OBJECT_NAME	NCHAR VARYING(128)	The name of the character set or collation.
OBJECT_TYPE	NCHAR VARYING(20)	One of: CHARACTER SET = the privilege is held on a character set COLLATION = the privilege is held on a collation DOMAIN = the privilege is held on a domain SEQUENCE = the privilege is held on a sequence.

Column name	Data type	Description
PRIVILEGE_TYPE	VARCHAR(20)	This will always be USAGE.
IS_GRANTABLE	VARCHAR(3)	One of: YES = the privilege is held WITH GRANT OPTION NO = the privilege is not held WITH GRANT OPTION.

INFORMATION_SCHEMA.USER_DEFINED_TYPES

The USER_DEFINED_TYPES system view shows the user-defined types defined and accessible by the current ident.

Column name	Data type	Description
USER_DEFINED_TYPE_CATALOG	NCHAR VARYING(128)	Catalog name for the user-defined type.
USER_DEFINED_TYPE_SCHEMA	NCHAR VARYING(128)	Schema name for the user-defined type.
USER_DEFINED_TYPE_NAME	NCHAR VARYING(128)	Name of the user-defined type.
USER_DEFINED_TYPE_CATEGORY	VARCHAR(20)	Contains STRUCTURED or DISTINCT.
IS_FINAL	VARCHAR(3)	YES if the user-defined type cannot have subtypes, otherwise NO.

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	Identifies the data type of the column. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
LOB_MAXIMUM_LENGTH	BIGINT	For the LOB data type, this shows the maximum length in bytes. For all other data types it is the null value.
CHARACTER_MAXIMUM_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in characters or bytes. For all other data types it is the null value.
CHARACTER_OCTET_LENGTH	INTEGER	For CHARACTER, LOB and BINARY data types, this shows the maximum length in octets. For all other data types it is the null value. For single octet character sets, this is the same as CHARACTER_MAX_LENGTH.
CHARACTER_SET_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the character set used by the column.
CHARACTER_SET_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the character set used by the column.

Column name	Data type	Description
CHARACTER_SET_NAME	NCHAR VARYING(128)	The name of the character set used by the column.
COLLATION_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the collation used by the column.
COLLATION_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the collation used by the column.
COLLATION_NAME	NCHAR VARYING(128)	The name of the collation used by the column.
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of significant digits allowed in the column. For all other data types it is the null value.
NUMERIC_PRECISION_RADIX	INTEGER	This shows whether the NUMERIC_PRECISION is given in a binary or decimal radix. The numeric radix is always decimal in Mimer SQL, therefore the value 10 is always shown for numeric data types. For all other data types it is the null value.
NUMERIC_SCALE	INTEGER	For NUMERIC and DECIMAL, this defines the total number of significant digits to the right of the decimal point. For BIGINT, INTEGER and SMALLINT, this is 0. For all other data types, it is the null value.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.

Column name	Data type	Description
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier. Can be one of: YEAR YEAR TO MONTH DAY HOUR MINUTE SECOND DAY TO HOUR DAY TO MINUTE DAY TO SECOND HOUR TO MINUTE HOUR TO SECOND MINUTE TO SECOND. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision (see *Interval Qualifiers* on page *55*). For other data types it is the null value.

INFORMATION_SCHEMA.VIEW_COLUMN_USAGE

The VIEW_COLUMN_USAGE system view lists the table columns on which views that are owned by the current ident depend.

Column name	Data type	Description
VIEW_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the view.
VIEW_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the view.
VIEW_NAME	NCHAR VARYING(128)	The name of the view.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.
COLUMN_NAME	NCHAR VARYING(128)	The name of the column.

INFORMATION_SCHEMA.VIEW_TABLE_USAGE

The VIEW_TABLE_USAGE system view lists the tables on which views that are owned by the current ident depend.

Column name	Data type	Description
VIEW_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the view.
VIEW_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the view.
VIEW_NAME	NCHAR VARYING(128)	The name of the view.
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table.
TABLE_NAME	NCHAR VARYING(128)	The name of the table.

INFORMATION_SCHEMA.VIEWS

The VIEWS system view lists the views to which the current ident as access.

Column name	Data type	Description
TABLE_CATALOG	NCHAR VARYING(128)	The name of the catalog containing the table or view.
TABLE_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the table or view.
TABLE_NAME	NCHAR VARYING(128)	The name of the table or view.
VIEW_DEFINITION	NCHAR VARYING(200)	The definition of the view as it would appear in a CREATE VIEW statement. If the actual definition would not fit into the maximum length of this column, the null value will be shown. In that case the definition can be found in INFORMATION_SCHEMA.EXT_SOURCE_DEFINITION.
CHECK_OPTION	VARCHAR(20)	The value CASCADED is shown if WITH CHECK OPTION was specified in the CREATE VIEW statement that created the view, and the value NONE is shown otherwise.

Column name	Data type	Description
IS_UPDATABLE	VARCHAR(3)	One of: YES = the view is updatable NO = the view is not updatable.

Standard Compliance

The table below summarizes standards compliance concerning the views in INFORMATION_SCHEMA.

Standard	Compliance	Comments
SQL-2016	Core	Fully compliant.
SQL-2016	Features outside core	Feature F231, "Privilege tables" support for the views TABLE_PRIVILEGES, COLUMN_PRIVILEGES and USAGE_PRIVILEGES. Feature T011, "TIME_STAMP domain in information_schema".
	Mimer SQL extension	All views starting with the name EXT_ are Mimer SQL extensions.

Appendix A

Reserved Words

The following keywords are reserved in Mimer SQL statements.
They must be enclosed in quotation marks if you want to use them as SQL identifiers.

ALL	ALLOCATE	ALTER
AND	ANY	AS
ASYMMETRIC	AT	ATOMIC
AUTHORIZATION	BEGIN	BETWEEN
BOTH	BY	CALL
CALLED	CASE	CAST
CHECK	CLOSE	COLLATE
COLUMN	COMMIT	CONDITION
CONNECT	CONSTRAINT	CORRESPONDING
CREATE	CROSS	CURRENT
CURRENT_DATE	CURRENT_PATH	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DAY	DEALLOCATE	DECLARE
DEFAULT	DELETE	DESCRIBE
DETERMINISTIC	DISCONNECT	DISTINCT
DO	DROP	ELSE
ELSEIF	END	ESCAPE
EXCEPT	EXECUTE	EXISTS
EXTERNAL	FALSE	FETCH
FIRST[a]	FOR	FOREIGN
FROM	FULL	FUNCTION
GET	GLOBAL	GRANT
GROUP	HANDLER	HAVING
HOLD	HOUR	IDENTITY

IF	IN	INDICATOR
INNER	INOUT	INSERT
INTERSECT	INTERVAL	INTO
IS	ITERATE	JOIN
LANGUAGE	LARGE	LEADING
LEAVE	LEFT	LIKE
LOCAL	LOCALTIME	LOCALTIMESTAMP
LOOP	MATCH	MEMBER
METHOD	MINUTE	MODIFIES
MODULE	MONTH	NATIONAL
NATURAL	NEW	NEXT[a]
NO	NOT	NULL
OF	OFFSET	OLD
ON	OPEN	OR
ORDER	OUT	OVERLAPS
PARAMETER	PRECISION	PREPARE
PRIMARY	PROCEDURE	READS
RECURSIVE	REFERENCES	REFERENCING
RELEASE	REPEAT	RESIGNAL
RESULT	RETURN	RETURNS
REVOKE	RIGHT	ROLLBACK
ROW	ROWS	SCROLL
SECOND	SELECT	SESSION_USER
SET	SIGNAL	SOME
SPECIFIC	SQL	SQLEXCEPTION
SQLSTATE	SQLWARNING	START
STATIC	SYMMETRIC	SYSTEM_USER
TABLE	THEN	TIMEZONE_HOUR
TIMEZONE_MINUTE	TO	TRAILING
TREAT	TRIGGER	TRUE
UNION	UNIQUE	UNKNOWN
UNTIL	UPDATE	USER
USING	VALUE	VALUES
VARYING	WHEN	WHERE

WHILE	WITH	
WITHOUT	YEAR	

a. Listed as non-reserved in the SQL-2016 standard.

Note: The keyword END-EXEC is also reserved specifically in statements embedded in COBOL programs.

Reserved Keywords in the SQL Standard

The following keywords are reserved in the SQL-2016 standard, but not in Mimer SQL:

ABS	ABSENT	ACOS
ARE	ARRAY	ARRAY_AGG
ARRAY_MAX_ CARDINALITY	ASENSITIVE	ASIN
ATAN	AVG	BEGIN_FRAME
BEGIN_PARTITION	BIGINT	BINARY
BLOB	BOOLEAN	CARDINALITY
CASCADED	CEIL	CEILING
CHAR	CHAR_LENGTH	CHARACTER
CHARACTER_LENGTH	CLASSIFIER	CLOB
COALESCE	COLLECT	CONTAINS
CONVERT	COPY	CORR
COS	COSH	COUNT
COVAR_POP	COVAR_SAMP	CUBE
CUME_DIST	CURRENT_CATALOG	CURRENT_DEFAULT_ TRANSFORM_GROUP
CURRENT_ROLE	CURRENT_ROW	CURRENT_SCHEMA
CURRENT_TRANSFORM _GROUP_FOR_TYPE	CYCLE	DATE
DEC	DECIMAL	DECFLOAT
DEFINE	DENSE_RANK	DEREF
DOUBLE	DYNAMIC	EACH
ELEMENT	EMPTY	END_FRAME
END_PARTITION	END-EXEC	EQUALS
EVERY	EXEC	EXP
EXTRACT	FILTER	FIRST_VALUE
FLOAT	FLOOR	FRAME_ROW

FREE	FUSION	GROUPING
GROUPS	INITIAL	INSENSITIVE
INT	INTEGER	INTERSECTION
JSON	JSON_ARRAY	JSON_ARRAYAGG
JSON_EXISTS	JSON_OBJECT	JSON_OBJECTAGG
JSON_QUERY	JSON_TABLE	JSON_TABLE_ PRIMITIVE
JSON_VALUE	LAG	LAST_VALUE
LATERAL	LEAD	LIKE_REGEX
LISTAGG	LN	LOG
LOG10	LOWER	MATCH_NUMBER
MATCH_RECOGNIZE	MATCHES	MAX
MERGE	MIN	MOD
MULTISET	NCHAR	NCLOB
NONE	NORMALIZE	NTH_VALUE
NTILE	NULLIF	NUMERIC
OCTET_LENGTH	OCCURRENCES_REGEX	OMIT
ONE	ONLY	OUTER
OVER	OVERLAY	PARTITION
PATTERN	PER	PERCENT
PERCENT_RANK	PERCENTILE_CONT	PERCENTILE_DISC
PERIOD	PORTION	POSITION
POSITION_REGEX	POWER	PRECEDES
PTF	RANGE	RANK
REAL	REF	REGR_AVGX
REGR_AVGY	REGR_COUNT	REGR_INTERCEPT
REGR_R2	REGR_SLOPE	REGR_SXX
REGR_SXY	REGR_SYY	ROLLUP
ROW_NUMBER	RUNNING	SAVEPOINT
SCOPE	SEARCH	SEEK
SENSITIVE	SHOW	SIMILAR
SIN	SINH	SKIP
SMALLINT	SPECIFICTYPE	SQL
SQRT	STDDEV_POP	STDDEV_SAMP

SUBMULTISET	SUBSET	SUBSTRING
SUBSTRING_REGEX	SUCCEEDS	SUM
SYSTEM	SYSTEM_TIME	TABLESAMPLE
TAN	TANH	TIME
TIMESTAMP	TRANSLATE	TRANSLATE_REGEX
TRANSLATION	TRIM	TRIM_ARRAY
TRUNCATE	UESCAPE	UNNEST
UPPER	VALUE_OF	VAR_POP
VAR_SAMP	VARBINARY	VARCHAR
VERSIONING	WHENEVER	WIDTH_BUCKET
WINDOW	WITHIN	

Appendix B

Character Sets

Character Data

For character data, the character set used by Mimer SQL is ISO 8859-1 (also known as Latin1). Character data is by default sorted in numerical order according to the `ISO8BIT` collation.

LOW \ HIGH	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			SP	0	@	P	`	p			NS	°	À	Ð	à	ð
1			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
2			"	2	B	R	b	r			¢	²	Â	Ò	â	ò
3			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
4			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
6			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
7			'	7	G	W	g	w			§	·	Ç	×	ç	÷
8			(	8	H	X	h	x			¨	¸	È	Ø	è	ø
9			)	9	I	Y	i	y			©	¹	É	Ù	é	ù
A			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
B			+	;	K	[	k	{			«	»	Ë	Û	ë	û
C			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü
D			-	=	M	]	m	}			SH	½	Í	Ý	í	ý
E			.	>	N	^	n	~			®	¾	Î	Þ	î	þ
F			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ

`SP` = space

`NS` = non-breaking space

`SH` = soft hyphen

National Character Data – Unicode

For national character data, Mimer SQL uses the Unicode character set. National character data is by default sorted in numerical order according to the `UCS_BASIC` collation.

For more information, see https://www.unicode.org.

Appendix C

Limits

LIMIT	Value
CHARACTER string length	15000
Variable length CHARACTER string	15000
NATIONAL CHARACTER string length	5000
Variable length NATIONAL CHARACTER string	5000
BINARY string length	15000
Variable BINARY string length	15000
DECIMAL precision in digits	45
FLOAT precision in digits	45
SMALLINT precision in bits	16
INTEGER precision in bits	32
BIGINT precision in bits	64
REAL precision in bits	24
DOUBLE PRECISION/FLOAT precision in bits	53
SQLCODE precision in digits	10
Fractional precision, in decimal digits, of TIME seconds component	9
Fractional precision, in decimal digits, of TIMESTAMP seconds component	9
Fractional precision, in decimal digits, of INTERVAL seconds component	9
Leading precision, in decimal digits, for INTERVAL data types	7-12
Fractional precision for INTERVAL data types that have a SECOND field	9
Number of columns in a table	252
Number of columns in an index	32
Number of tables referenced in a statement	unlimited [a]
Number of cursors open simultaneously	unlimited [b]

LIMIT	Value
Number of columns a single update statement can update	unlimited [a]
Number of parameters for a function or procedure	unlimited [b]
SQL statement length in bytes	unlimited [b]
Total length of a row, the sum of: Two bytes for column identification Lengths of all character/binary columns Precisions of all numeric columns	32000
Identifier length	128

a. There is a limit on the total complexity of an SQL statement.
b. The limit is only dependent on the amount of virtual memory available to the application.

Appendix D

Deprecated Features

Some non-standard features of earlier versions of Mimer SQL are retained for backward compatibility. Where these features have equivalents in the standard implementation, only the standard form is documented in the main body of this manual. Non-standard forms are documented below. Use of the standard forms is strongly recommended in new applications.

Non-standard Mimer SQL features which do not have equivalents in the standards are documented in the main body of the manual.

Indicator Variables

Host variables with negative indicator variables are assigned null. In previous versions of Mimer SQL, the indicator variable could be given any negative value. The accepted standards however define the value of `-1` as an indicator for null and reserve other negative values for future use. Use of values other than `-1` is therefore not recommended.

Operators

Operator	Standard form	Alternative form(s)
string concatenation	`||`	
not equal to	<>	!=
less than or equal to	<=	
greater than or equal to	>=	

Statements

ALTER IDENT Change Password

The `ALTER IDENT USING` syntax and the `ALTER IDENT IDENTIFIED BY` syntax to change password are deprecated and have been replaced by the `SET PASSWORD` syntax. The old syntax is still supported for backward compatibility.

CREATE IDENT AS OS_USER

The possibility to specify `OS_USER` as ident type is deprecated. The syntax is still supported for backward compatibility.

GET DIAGNOSTICS EXCEPTION INFO

The SQL-2008 standard has adjusted the GET DIAGNOSTICS syntax. The keyword `EXCEPTION` has been replaced with the keyword `CONDITION`. As a consequence the `exception-clause` is now called `condition-clause`.

The old `GET DIAGNOSTICS EXCEPTION INFO` syntax is still supported for backward compatibility.

JOIN Without SELECT

The way to write a join without starting with the `SELECT` keyword was removed from the SQL standard in 2003, and has also been removed from the Mimer SQL syntax.

So instead of just writing

```
table1 INNER JOIN table2 ON table1.col1 = table2.co2
```

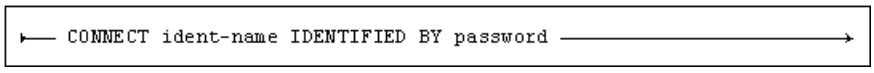
specify the `SELECT` keyword, like:

```
SELECT * FROM table1 INNER JOIN table2 ON table1.col1 = table2.co2
```

CONNECT

The syntax of the standard `CONNECT` statement differs from that in earlier versions of Mimer SQL. The previous form is still supported for backward compatibility.

Backward compatibility syntax:



Note: The `CONNECT` statement and the standard-compliant `CONNECT TO` statement have different default modes for `SET TRANSACTION START`. `CONNECT` uses `SET TRANSACTION START EXPLICIT`, while `CONNECT TO` uses `SET TRANSACTION START IMPLICIT`.

ORDER BY Ordinal Position

Using an integer value to represent the ordinal position in an `ORDER BY` specification is deprecated due to changes in the SQL standards.

SELECT NULL

The use of the keyword `NULL` in a select list is still supported for backward compatibility but should be regarded as a deprecated feature in Mimer SQL.

Instead, specify `CAST(NULL AS data-type)`.

Example

```
SELECT c1, int_col FROM t1
UNION
SELECT c1, CAST(NULL AS integer) FROM t2;
```

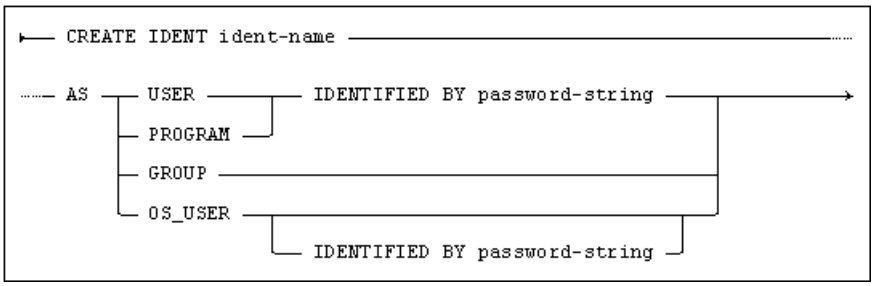
SET TRANSACTION CHANGES

Following the introduction of the `SET TRANSACTION READ` and `SET TRANSACTION ISOLATION LEVEL` options, the `SET TRANSACTION CHANGES` options are no longer supported. The options are currently supported for backward compatibility, but will be removed in the next version of Mimer SQL.

CREATE IDENT

The keyword `USING` is now used when specifying the `password-string` for the ident being created. The use of `IDENTIFIED BY` is deprecated.

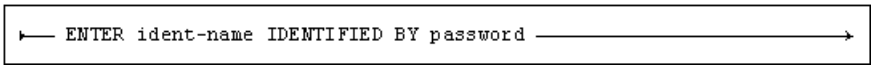
The following syntax is supported for backward compatibility only:



ENTER

The keyword `USING` is now used when specifying the `password` for the program ident being entered. The use of `IDENTIFIED BY` is deprecated.

The following syntax is supported for backward compatibility only:



Program Idents

MIMER_SW

The right to create and manage databank shadows was previously granted by having `EXECUTE` on the program ident `MIMER_SW`.

This has now been replaced by the `SHADOW` system privilege.

MIMER_BR

The right to perform backup and restore operations was previously granted by having `EXECUTE` on the program ident `MIMER_BR`.

This has now been replaced by the `BACKUP` system privilege.

MIMER_SC

The right to execute the UPDATE STATISTICS statement was previously granted by having EXECUTE on the program ident MIMER_SC.

This has now been replaced by the STATISTICS system privilege.

Functions

The following functions have been deprecated in the current version of Mimer SQL.

BIT_LENGTH

The BIT_LENGTH function is no longer supported. The function OCTET_LENGTH's return value multiplied by 8 can be used as an alternative.

See *OCTET_LENGTH on page 112*.

Arithmetic Functions

The functions DACOS, DASIN, DATAN, DATAN2, DCOS, DCOT, DDEGREES, DEXP, DLOG, DLOG10, DPOWER, DRADIANS, DSIN, DSQRT and DTAN are no longer supported.

In Mimer version 11 the following functions are supported: ACOS, ASIN, ATAN, ATAN2, COS, COSH, COT, EXP, LN, LOG10, POWER, SIN, SINH, SQRT, TAN and TANH.

Datetime Scalar Functions

The following scalar functions have been deprecated in the current version of Mimer SQL. They will be re-introduced in a future version with changed functionality.

CURRENT_TIME

This function, in its present form, has been removed since it's deprecated.

The same functionality is now provided by the LOCALTIME function.

CURRENT_TIMESTAMP

This function, in its present form, has been removed since it's deprecated.

The same functionality is now provided by the LOCALTIMESTAMP function.

Data Dictionary Views

The MIMER and INFO_SCHEM predefined system views on the data dictionary tables, that were supported in previous versions of Mimer SQL have now been replaced by the standard system views belonging to the schema INFORMATION_SCHEMA, which is owned by ident SYSTEM.

The FIPS_DOCUMENTATION system views have been removed.

Host Variable Types

The following SQL descriptor host variable types are deprecated:

Host variable	TYPE
MIMER_BINARY	-13

Appendix E

Return Status and Conditions

In Mimer SQL, status information is returned in the SQLCODE and SQLSTATE variables.

The SQLSTATE variable must be declared as a 5-character long string (excluding any terminating null byte), and the declaration of SQLSTATE must be made between BEGIN DECLARE SECTION and END DECLARE SECTION. The SQLSTATE variable together with the GET DIAGNOSTICS statement will provide the application with information about the most recently executed SQL statement.

The SQLCODE variable must be declared as integer. If neither an SQLSTATE nor an SQLCODE variable is declared between BEGIN DECLARE SECTION and END DECLARE SECTION, Mimer SQL will assume the existence of an SQLCODE variable. (I.e. declared somewhere outside the declare section, but still within scope.)

There are three different types of conditions in Mimer SQL; NOT FOUND, SQLEXCEPTION and SQLWARNING (see the *WHENEVER* on page *434*).

The NOT FOUND condition is returned by setting SQLSTATE to No data ('02000'), or by setting SQLCODE to +100. This is referred to as "a NOT FOUND condition code is returned" in this manual.

SQLEXCEPTION's are returned using an error code in SQLSTATE, or a negative value in SQLCODE. This is referred to as "an error code is returned" in this manual.

SQLWARNING's are returned by setting either a Success with warning -code in SQLSTATE, or a value > 0 in SQLCODE. This is referred to as "a warning flag is set" in this manual.

More detailed information about certain operations can be obtained by the GET DIAGNOSTICS statement described in *GET DIAGNOSTICS* on page *351*.

SQLSTATE Return Codes

SQLSTATE contains a 5 character long return code string that indicates status of an SQL statement. These return codes follows the established standards. Observe that not all standardized SQLSTATE return codes are used by Mimer SQL.

The SQLSTATE values consists of two fields: the class, which is the first two characters of the string, and the subclass, which is the terminating three characters of the string.

For a full list of SQLSTATE values returned by Mimer SQL, see *Mimer SQL Programmer's Manual, Appendix B, List of SQLSTATE Values*.

SQLCODE Return Codes

The values of `SQLCODE` are the same as the values for the native Mimer SQL return codes, as described in the *Mimer SQL Programmer's Manual, Appendix B, Native Mimer SQL Return Codes*.

Appendix F

SQL-2016 Compliance

The ANSI/ISO SQL-2016 standard is divided into a number of named features. In this appendix it is shown which of these features that are supported in Mimer SQL. The information can also be retrieved from the data dictionary by selecting from the view `INFORMATION_SCHEMA.SQL_FEATURES`.

SQL-2016 Core Features

Feature ID	Feature name	Supported
B012	Embedded C	Yes
B013	Embedded COBOL	Yes
B014	Embedded Fortran	Yes
E011	Numeric data types	Yes
E011-01	INTEGER and SMALLINT data types (including all spellings)	Yes
E011-02	REAL, DOUBLE PRECISON, and FLOAT data types	Yes
E011-03	DECIMAL and NUMERIC data types	Yes
E011-04	Arithmetic operators	Yes
E011-05	Numeric comparison	Yes
E011-06	Implicit casting among the numeric data types	Yes
E021	Character string types	Yes
E021-01	CHARACTER data type (including all its spellings)	Yes
E021-02	CHARACTER VARYING data type (including all its spellings)	Yes

E021-03	Character literals	Yes
E021-04	CHARACTER_LENGTH function	Yes
E021-05	OCTET_LENGTH function	Yes
E021-06	SUBSTRING function	Yes
E021-07	Character concatenation	Yes
E021-08	UPPER and LOWER functions	Yes
E021-09	TRIM function	Yes
E021-10	Implicit casting among the fixed-length and variable-length character string types	Yes
E021-11	POSITION function	Yes
E021-12	Character comparison	Yes
E031	Identifiers	Yes
E031-01	Delimited identifiers	Yes
E031-02	Lower case identifiers	Yes
E031-03	Trailing underscore	Yes
E051	Basic query specification	Yes
E051-01	SELECT DISTINCT	Yes
E051-02	GROUP BY clause	Yes
E051-04	GROUP BY can contain columns not in select-list	Yes
E051-05	Select list items can be renamed	Yes
E051-06	HAVING clause	Yes
E051-07	Qualified * in select list	Yes
E051-08	Correlation names in the FROM clause	Yes
E051-09	Rename columns in the FROM clause	Yes
E061	Basic predicates and search conditions	Yes
E061-01	Comparison predicate	Yes
E061-02	BETWEEN predicate	Yes
E061-03	IN predicate with list of values	Yes
E061-04	LIKE predicate	Yes
E061-05	LIKE predicate: ESCAPE clause	Yes
E061-06	NULL predicate	Yes
E061-07	Quantified comparison predicate	Yes

E061-08	EXISTS predicate	Yes
E061-09	Subqueries in comparison predicate	Yes
E061-11	Subqueries in IN predicate	Yes
E061-12	Subqueries in quantified comparison predicate	Yes
E061-13	Correlated subqueries	Yes
E061-14	Search condition	Yes
E071	Basic query expressions	Yes
E071-01	UNION DISTINCT table operator	Yes
E071-02	UNION ALL table operator	Yes
E071-03	EXCEPT DISTINCT table operator	Yes
E071-05	Columns combined via table operators need not have exactly the same data type	Yes
E071-06	Table operators in subqueries	Yes
E081	Basic Privileges	Yes
E081-01	SELECT privilege at the table level	Yes
E081-02	DELETE privilege	Yes
E081-03	INSERT privilege at the table level	Yes
E081-04	UPDATE privilege at the table level	Yes
E081-05	UPDATE privilege at the column level	Yes
E081-06	REFERENCES privilege at the table level	Yes
E081-07	REFERENCES privilege at the column level	Yes
E081-08	WITH GRANT OPTION	Yes
E081-09	USAGE privilege	Yes
E081-10	EXECUTE privilege	Yes
E091	Set functions	Yes
E091-01	AVG	Yes
E091-02	COUNT	Yes
E091-03	MAX	Yes
E091-04	MIN	Yes
E091-05	SUM	Yes
E091-06	ALL quantifier	Yes
E091-07	DISTINCT qualifier	Yes

E101	Basic data manipulation	Yes
E101-01	INSERT statement	Yes
E101-03	Searched UPDATE statement	Yes
E101-04	Searched DELETE statement	Yes
E111	Single row SELECT statement	Yes
E121	Basic cursor support	Yes
E121-01	DECLARE CURSOR	Yes
E121-02	ORDER BY columns need not be in select list	Yes
E121-03	Value expressions in ORDER BY clause	Yes
E121-04	Open statement	Yes
E121-06	Positioned UPDATE statement	Yes
E121-07	Positioned DELETE statement	Yes
E121-08	CLOSE statement	Yes
E121-10	FETCH statement: implicit NEXT	Yes
E121-17	WITH HOLD cursors	Yes
E131	Null value support (nulls in lieu of values)	Yes
E141	Basic integrity constraints	Yes
E141-01	NOT NULL constraints	Yes
E141-02	UNIQUE constraints of NOT NULL columns	Yes
E141-03	PRIMARY KEY constraints	Yes
E141-04	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	Yes
E141-06	CHECK constraints	Yes
E141-07	Column defaults	Yes
E141-08	NOT NULL inferred on PRIMARY KEY	Yes
E141-10	Names in a foreign key can be specified in any order	Yes
E151	Transaction support	Yes
E151-01	COMMIT statement	Yes
E151-02	ROLLBACK statement	Yes
E152	Basic SET TRANSACTION statement	Yes

E152-01	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	Yes
E152-02	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	Yes
E153	Updatable queries with subqueries	Yes
E161	SQL comments using leading double minus	Yes
E171	SQLSTATE support	Yes
F031	Basic schema manipulation	Yes
F031-01	CREATE TABLE statement to create persistent base tables	Yes
F031-02	CREATE VIEW statement	Yes
F031-03	GRANT statement	Yes
F031-04	ALTER TABLE statement: ADD COLUMN clause	Yes
F031-13	DROP TABLE statement: RESTRICT clause	Yes
F031-16	DROP VIEW statement: RESTRICT clause	Yes
F031-19	REVOKE statement: RESTRICT clause	Yes
F041	Basic joined tables	Yes
F041-01	Inner join (but not necessarily the INNER keyword)	Yes
F041-02	INNER keyword	Yes
F041-03	LEFT OUTER JOIN	Yes
F041-04	RIGHT OUTER JOIN	Yes
F041-05	Outer joins can be nested	Yes
F041-07	The inner table in a left or right outer join can also be used in an inner join	Yes
F041-08	All comparison operators are supported (rather than just =)	Yes
F051	Basic date and time	Yes
F051-01	DATE data type (including DATE literal)	Yes
F051-02	TIME data type (including TIME literal) with fractional seconds precision of 0	Yes
F051-03	TIMESTAMP data type (including TIMESTAMP literal) with fractional seconds precision of 0 and 6	Yes

F051-04	Comparison predicate on DATE, TIME, and TIMESTAMP data types	Yes
F051-05	Explicit CAST between datetime types and character types	Yes
F051-06	CURRENT_DATE	Yes
F051-07	LOCALTIME	Yes
F051-08	LOCALTIMESTAMP	Yes
F081	UNION and EXCEPT in views	No
F131	Grouped operations	Yes
F131-01	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	Yes
F131-02	Multiple tables supported in queries with grouped views	Yes
F131-03	Set functions supported in queries with grouped views	Yes
F131-04	Subqueries with GROUP BY and HAVING clauses and grouped views	Yes
F131-05	Single row SELECT with GROUP BY and HAVING clauses and grouped views	Yes
F181	Multiple module support	Yes
F201	CAST function	Yes
F221	Explicit defaults	Yes
F261	CASE expression	Yes
F261-01	Simple CASE	Yes
F261-02	Searched CASE	Yes
F261-03	NULLIF	Yes
F261-04	COALESCE	Yes
F311	Schema definition statement	Yes
F311-01	CREATE SCHEMA	Yes
F311-02	CREATE TABLE for persistent base tables	Yes
F311-03	CREATE VIEW	Yes
F311-04	CREATE VIEW: WITH CHECK OPTION	Yes
F311-05	GRANT statement	Yes
F471	Scalar subquery values	Yes

F481	Expanded NULL predicate	Yes
F812	Basic flagging	Yes
S011	Distinct data types	Yes
T321	Basic SQL invoked routines	Yes
T321-01	User-defined functions with no overloading	Yes
T321-02	User-defined stored procedures with no overloading	Yes
T321-03	Function invocation	Yes
T321-04	CALL statement	Yes
T321-05	RETURN statement	Yes

Features Outside Core Supported by Mimer SQL

Feature ID	Feature Name
B021	Direct SQL
B031	Basic dynamic SQL
B032	Extended dynamic SQL
B032-01	<describe input> statement
B033	Untyped SQL invoked function argument
B112	Module language C
B113	Module language COBOL
B114	Module language Fortran
B128	Routine language SQL
F032	CASCADE drop behavior
F033	ALTER TABLE statement: DROP COLUMN clause
F034	Extended REVOKE statement
F052	Intervals and datetime arithmetic
F053	OVERLAPS predicate
F054	Timestamp in DATE type precedence list
F111	Isolation levels other than SERIALIZABLE
F111-01	READ UNCOMMITTED isolation level
F111-02	READ COMMITTED isolation level
F111-03	REPEATABLE READ isolation level

F121	Basic diagnostics management
F121-01	GET DIAGNOSTICS statement
F121-02	SET TRANSACTION statement: DIAGNOSTICS SIZE clause
F122	Enhanced diagnostics management
F171	Multiple schemas per user
F191	Referential delete actions
F222	INSERT statement: DEFAULT VALUES clause
F231	Privilege tables
F251	Domain support
F271	Compound character literals
F281	LIKE enhancements
F291	UNIQUE predicate
F302	INTERSECT table operator
F302-01	INTERSECT DISTINCT table operator
F302-02	INTERSECT ALL table operator
F304	EXCEPT ALL table operator
F341	Usage tables
F361	Subprogram support
F381	Extended schema manipulation
F381-01	ALTER TABLE statement: ALTER COLUMN clause
F381-02	ALTER TABLE statement: ADD CONSTRAINT clause
F381-03	ALTER TABLE statement: DROP CONSTRAINT clause
F382	Alter column data type
F391	Long identifiers
F392	Unicode escapes in identifiers
F393	Unicode escapes in literals
F401-01	NATURAL JOIN
F401-04	CROSS JOIN
F421	National character
F431	Read-only scrollable cursors
F441	Extended set function support
F442	Mixed column references in set functions

F491	Constraint management
F555	Enhanced seconds precision
F561	Full value expressions
F571	Truth value tests
F591	Derived tables
F661	Simple tables
F672	Retrospective check constraints
F673	Reads SQL-data routine invocations in CHECK constraints
F690	Collation support
F692	Enhanced collation support
F701	Referential update actions
F721	Deferrable constraints
F731	INSERT column privileges
F771	Connection management
F781	Self-referencing operations
F850	Top-level <order by clause> in <query expression>
F851	<order by clause> in subqueries
F852	Top-level <order by clause> in views
F855	Nested <order by clause> in <query expression>
F856	Nested <fetch first clause> in <query expression>
F857	Top-level <fetch first clause> in <query expression>
F858	<fetch first clause> in subqueries
F859	Top-level <fetch first clause> in views
F860	dynamic <fetch first row count> in <fetch first clause>
F861	Top-level <result offset clause> in <query expression>
F862	<result offset clause> in subqueries
F863	Nested <result offset clause> in <query expression>
F864	Top-level <result offset clause> in views
F865	dynamic <offset row count> in <result offset clause>
P001	Stored Modules
P002	Computational completeness
P003	Information Schema views

P004	Extended CASE statement
P005	Qualified SQL variable references
P006	Multiple assignment
S028	Permutable UDT options list
T011	Timestamp in Information Schema
T021	BINARY and VARBINARY data types
T022	Advanced support for BINARY and VARBINARY data types
T023	Compound binary literals
T024	Spaces in binary literals
T031	BOOLEAN data type
T041	Basic LOB data type support. Mimer SQL does not support the following sub-features: T041-03 “POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types” (except SUBSTRING that is supported) T041-04 “Concatenation of LOB data types” T041-05 “LOB locator: non-holdable”.
T071	BIGINT data type
T101	Enhanced nullability determination
T121	WITH (excluding RECURSIVE) in query expression
T122	WITH (excluding RECURSIVE) in subquery
T132	Recursive query in subquery
T151	DISTINCT predicate
T152	DISTINCT predicate with negation
T176	Sequence generator support
T177	Sequence generator support: simple restart option
T191	Referential action RESTRICT
T211	Basic trigger capability
T212	Enhanced trigger capability
T213	INSTEAD OF triggers
T241	START TRANSACTION statement
T285	Enhanced derived column names
T312	OVERLAY function
T321	Basic SQL-invoked routines

T341	Overloading of SQL-invoked functions and SQL-invoked procedures
T441	ABS and MOD functions
T461	Symmetric BETWEEN predicate
T501	Enhanced EXISTS predicate
T551	Optional key words for default syntax
T591	UNIQUE constraints of possibly null columns
T622	Trigonometric functions
T624	Common logarithm functions
T631	IN predicate with one list element
T641	Multiple column assignment
T655	Cyclically dependent routines
P001	Stored modules
P002	Computational completeness
P003	Information Schema views

Appendix G

Languages

Mimer SQL contains a large number of collations for different languages:

- Afrikaans
- Albanian
- Arabic
- Armenian
- Arumanian
- Assamese
- Asturian
- Azerbaijani
- Basque
- Belarusian
- Bengali
- Bosnian
- Breton
- Bulgarian
- Catalan
- Chinese KangXi
- Chinese PinYin
- Chinese_Pinyin_Name
- Chinese Stroke
- Chinese ZhuYin
- Chinese_Zhuyin_Name
- Corsican
- Croatian
- Czech
- Danish
- Dutch
- Dzongkha
- English
- Esperanto

- Estonian
- Faroese
- Filipino
- Finnish
- French
- Frisian
- Friulian
- Galician
- Georgian
- German
- German Phonebook
- Greek
- Greenlandic
- Gujarati
- Hausa
- Hebrew
- Hindi
- Hungarian
- Icelandic
- Igbo
- Indonesian
- Irish Gaelic
- Italian
- Japanese
- Javanese
- Kannada
- Kazakh
- Khmer
- Kirghiz
- Korean
- Kurdish
- Lao
- Latin
- Latvian
- Lithuanian
- Luxembourgish
- Macedonian
- Malay

- Malayalam
- Maltese
- Marathi
- Moldavian
- Myanmar
- Nepali
- Norwegian
- Occitan
- Oriya
- Persian
- Polish
- Portuguese
- Punjabi
- Romanian
- Romansch
- Russian
- Sami (Inari, Lule, Northern, Skolt, Southern)
- Sanskrit
- Scots
- Scottish Gaelic
- Serbian
- Sesotho
- Sinhala
- Slovak
- Slovenian
- Sorbian Lower
- Sorbian Upper
- Spanish
- Spanish Traditional
- Swedish
- Tamil
- Tatar
- Telugu
- Thai
- Tibetan
- Turkish
- Turkmen
- Ukrainian

- Urdu
- Uzbek
- Vietnamese
- Welsh
- Yiddish
- Yoruba
- Zulu

All collations available are found in the `INFORMATION_SCHEMA.COLLATIONS` view. Mimer SQL's predefined level 1 collations have names ending with `_1`, e.g. `ENGLISH_1`, the predefined level 2 collations have names ending with `_2`, e.g. `ENGLISH_2`, and the predefined level 3 collations have names ending with `_3`, e.g. `ENGLISH_3`.

Note: Mimer Information Technology AB reserves the right to change the contents of these collations in future releases.

Appendix H

Type Precedence Lists

The following table describes the parameter overloading type precedence lists:

Data type	Precedence list
CHARACTER VARYING	CHARACTER VARYING CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHARACTER
CHARACTER	CHARACTER CHARACTER VARYING NATIONAL CHARACTER NATIONAL CHARACTER VARYING
NATIONAL CHARACTER VARYING	NATIONAL CHARACTER VARYING NATIONAL CHARACTER CHARACTER VARYING CHARACTER
NATIONAL CHARACTER	NATIONAL CHARACTER NATIONAL CHARACTER VARYING CHARACTER CHARACTER VARYING
BINARY	BINARY BINARY VARYING
BINARY VARYING	BINARY VARYING BINARY
INTERVAL YEAR	INTERVAL YEAR INTERVAL YEAR to MONTH INTERVAL MONTH
INTERVAL YEAR to MONTH	INTERVAL YEAR to MONTH INTERVAL YEAR INTERVAL MONTH

Data type	Precedence list
INTERVAL MONTH	INTERVAL MONTH INTERVAL YEAR to MONTH INTERVAL YEAR
INTERVAL DAY	INTERVAL DAY INTERVAL DAY to HOUR INTERVAL DAY to MINUTE INTERVAL DAY to SECOND INTERVAL HOUR INTERVAL HOR to MINUTE INTERVAL HOUR to SECOND INTERVAL MINUTE INTERVAL MINUTE to SECOND INTERVAL SECOND
INTERVAL DAY to HOUR	INTERVAL DAY to HOUR INTERVAL DAY to MINUTE INTERVAL DAY to SECOND INTERVAL DAY INTERVAL HOUR INTERVAL HOR to MINUTE INTERVAL HOUR to SECOND INTERVAL MINUTE INTERVAL MINUTE to SECOND INTERVAL SECOND
INTERVAL DAY to MINUTE	INTERVAL DAY to MINUTE INTERVAL DAY to SECOND INTERVAL DAY to HOUR INTERVAL DAY INTERVAL HOR to MINUTE INTERVAL HOUR to SECOND INTERVAL HOUR INTERVAL MINUTE INTERVAL MINUTE to SECOND INTERVAL SECOND
INTERVAL DAY to SECOND	INTERVAL DAY to SECOND INTERVAL DAY to MINUTE INTERVAL DAY to HOUR INTERVAL DAY INTERVAL HOUR to SECOND INTERVAL HOUR to MINUTE INTERVAL HOUR INTERVAL MINUTE to SECOND INTERVAL MINUTE INTERVAL SECOND

Data type	Precedence list
INTERVAL HOUR	INTERVAL HOUR INTERVAL HOUR to MINUTE INTERVAL HOUR to SECOND INTERVAL DAY to HOUR INTERVAL DAY to MINUTE INTERVAL DAY to SECOND INTERVAL DAY INTERVAL MINUTE INTERVAL MINUTE to SECOND INTERVAL SECOND
INTERVAL HOUR to MINUTE	INTERVAL HOUR to MINUTE INTERVAL HOUR to SECOND INTERVAL HOUR INTERVAL DAY to MINUTE INTERVAL DAY to SECOND INTERVAL DAY to HOUR INTERVAL DAY INTERVAL MINUTE INTERVAL MINUTE to SECOND INTERVAL SECOND
INTERVAL HOUR to SECOND	INTERVAL HOUR to SECOND INTERVAL HOUR to MINUTE INTERVAL HOUR INTERVAL DAY to SECOND INTERVAL DAY to MINUTE INTERVAL DAY to HOUR INTERVAL DAYI NTERVAL MINUTE to SECOND INTERVAL MINUTE INTERVAL SECOND
INTERVAL MINUTE	INTERVAL MINUTE INTERVAL MINUTE to SECOND INTERVAL HOUR to MINUTE INTERVAL HOUR to SECOND INTERVAL HOUR INTERVAL DAY to MINUTE INTERVAL DAY to HOUR INTERVAL DAY INTERVAL DAY to SECOND INTERVAL SECOND

Data type	Precedence list
INTERVAL MINUTE to SECOND	INTERVAL MINUTE to SECOND INTERVAL HOUR to SECOND INTERVAL DAY to SECOND INTERVAL MINUTE INTERVAL HOUR to MINUTE INTERVAL DAY to MINUTE INTERVAL HOUR INTERVAL DAY to HOUR INTERVAL DAY INTERVAL SECOND
INTERVAL SECOND	INTERVAL SECOND INTERVAL MINUTE to SECOND INTERVAL HOUR to SECOND INTERVAL DAY to SECOND INTERVAL MINUTE INTERVAL HOUR to MINUTE INTERVAL HOUR INTERVAL DAY to MINUTE INTERVAL DAY to HOUR INTERVAL DAY
DATE	DATE
TIME	TIME
TIMESTAMP	TIMESTAMP
SMALLINT	SMALLINT Integer(p) INTEGER BIGINT DECIMAL NUMERIC Float(p) REAL FLOAT DOUBLE PRECISION
Integer(p)	Integer(p) INTEGER BIGINT DECIMAL NUMERIC Float(p) REAL FLOAT DOUBLE PRECISION SMALLINT

Data type	Precedence list
INTEGER	INTEGER BIGINT DECIMAL NUMERIC Float(p) REAL FLOAT DOUBLE PRECISION Integer(p) SMALLINT
BIGINT	BIGINT DECIMAL NUMERIC Float(p) REAL FLOAT DOUBLE PRECISION Integer(p) INTEGER SMALLINT
DECIMAL	DECIMAL NUMERIC Float(p) REAL FLOAT DOUBLE PRECISION BIGINT INTEGER Integer(p) SMALLINT
NUMERIC	NUMERIC DECIMAL Float(p) REAL FLOAT DOUBLE PRECISION BIGINT INTEGER Integer(p) SMALLINT

Data type	Precedence list
Float(p)	Float(p) REAL FLOAT DOUBLE PRECISION DECIMAL NUMERIC BIGINT INTEGER Integer(p) SMALLINT
REAL	REAL FLOAT DOUBLE PRECISION Float(p) DECIMAL NUMERIC BIGINT INTEGER Integer(p) SMALLINT
FLOAT	FLOAT DOUBLE PRECISION REAL Float(p) DECIMAL NUMERIC BIGINT INTEGER Integer(p) SMALLINT
DOUBLE PRECISION	DOUBLE PRECISION FLOAT REAL Float(p) DECIMAL NUMERIC BIGINT INTEGER Integer(p) SMALLINT
BOOLEAN	BOOLEAN

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Mimer SQL

Programmer's Manual

Version 11.0

Mimer SQL, Programmer's Manual, Version 11.0, December 2024
© Copyright Mimer Information Technology AB.

The contents of this manual may be printed in limited quantities for use at a Mimer SQL installation site. No parts of the manual may be reproduced for sale to a third party.
Information in this document is subject to change without notice. All registered names, product names and trademarks of other companies mentioned in this documentation are used for identification purposes only and are acknowledged as the property of the respective company. Companies, names and data used in examples herein are fictitious unless otherwise noted.

Produced and published by Mimer Information Technology AB, Uppsala, Sweden.

Mimer SQL Web Sites:
https://developer.mimer.com
https://www.mimer.com

Contents

Chapter 1

Introduction

Mimer SQL is an advanced relational database management system (RDBMS) developed by Mimer Information Technology AB.

The main characteristics of Mimer SQL are zero maintenance, small footprint and high performance. These are based on a number of unique technical solutions to handle some of the more complicated functionality that a database management system must provide.

For example, Mimer SQL provides a solution to the problem of allowing simultaneous access to the database without the danger of a deadlock occurring. This greatly simplifies database management and allows truly scalable performance, even during heavy system-load.

Another significant technical innovation is the data storage mechanism, which is constantly optimized for the highest possible performance and ensures that no manual reorganization of the database is ever needed.

Mimer SQL offers a uniquely scalable and portable solution, including multi-core support. The product is available on a wide range of platforms from small embedded and handheld devices running for example Android or Linux, to workgroup and enterprise servers running Linux, Windows, macOS and OpenVMS. This makes Mimer SQL ideally suited for open environments where interoperability, portability and small footprint are important.

The database management language Mimer SQL (Structured Query Language) is compatible in all essential features with the currently accepted SQL standards, see the *Mimer SQL Reference Manual, Chapter 3, Introduction to SQL Standards*, for details.

About this Manual

The manual is intended for application developers working with Mimer SQL.

This manual describes the usage of SQL in application programs, and provides, together with the *Mimer SQL Reference Manual*, the complete reference material for Mimer SQL.

For details on how to read the syntax diagrams that appear in this manual see the *Mimer SQL Reference Manual, Chapter 2, Reading SQL Syntax Diagrams*.

This manual describes how SQL statements may be embedded in application programs written in conventional host languages. It also describes how to create and use stored procedures and triggers.

The information contained in this manual generally applies to all the platforms supported by Mimer SQL. From time to time platform-specific notes appear in the general description, presented as follows:

Linux:	Denotes information that applies specifically to Linux and macOS platforms.

VMS:	Denotes information that applies specifically to OpenVMS platforms.

Win:	Denotes information that applies specifically to Windows platforms.

Database APIs

You can access Mimer SQL using the following native database application interfaces:

- **ADO.NET**

  ADO.NET is the interface of choice when developing database applications in the Microsoft .NET framework and it is the natural successor for ADO (ActiveX Data Objects).

  The Mimer Data Provider is used to connect to Mimer SQL from .NET. How it works is described in https://developer.mimer.com/article/ado-net/.

  The Mimer Data Provider is not included in the regular Mimer SQL distribution, instead the latest release can be downloaded from https://developer.mimer.com/downloads.

- **JDBC**

  JDBC™ is a Java database API. Through JDBC, Mimer SQL can support many JDBC based tools.

- **ODBC**

  ODBC is a database independent interface. Through ODBC, Mimer SQL can support many ODBC based tools.

- **Embedded SQL (ESQL)**

  ESQL is used through a host programming language (C/C++, COBOL or Fortran as available on the host computer). SQL statements are included as part of the source code for an application program, which is compiled and linked with the appropriate language-specific facilities. The SQL statements are executed in the context of the application program.

- **Module SQL (MSQL)**

  MSQL enables you to call SQL statements in a host program written in C/C++, COBOL, Fortran or Pascal, without embedding the actual SQL statements in the host program. The SQL statements are explicitly put into a separate SQL module, that is written in the Module language and maintained separately from the host program.

- **Mimer SQL C API**

  The Mimer SQL C API is a native C library suitable for tool integration and application development in environments where API standardization is not a requirement. The following characteristics describe the API:

  - Simplicity

  - Platform independence
  - Small footprint
  - Tight fit with the Mimer SQL application/database communication model.

- **Mimer SQL Micro C API**

  The Mimer SQL Micro C API is mainly targeted for memory and CPU constrained environments.

In addition, there are some additional interfaces supported by Mimer SQL, see https://developer.mimer.com/doc/database-apis/.

Prerequisites

Application developers using this manual are assumed to have a working acquaintance with the principles of the relational database model in general and of Mimer SQL in particular.

Knowledge of Mimer SQL is of course an advantage, although experience with other standard-compliant SQL implementations will suffice. Experience of Mimer SQL is best gained through interactive use of DbVisualizer or BSQL, both included in the Mimer SQL distribution.

Related Mimer SQL Publications

- **Mimer SQL Reference Manual**

  contains a complete description of the syntax and usage of all statements in Mimer SQL and is a necessary complement to this manual.

- **Mimer SQL User's Manual**

  contains a description of the BSQL facilities. A user-oriented guide to the SQL statements is also included, which may provide help for less experienced users in formulating statements correctly (particularly the SELECT statement, which can be quite complex).

- **Mimer SQL System Management Handbook**

  describes system administration functions, including export/import, backup/restore, databank shadowing and the statistics functionality.

  The information in that manual is used primarily by the system administrator, and is not required by application program developers. The SQL statements that are part of the System Management API are described in the *Mimer SQL Reference Manual*.

- **Mimer SQL Platform-specific documents**

  contain platform-specific information. A set of one or more documents is provided, where required, for each platform on which Mimer SQL is supplied.

- **Mimer SQL Release Notes**

  contain general and platform-specific information relating to the Mimer SQL release for which they are supplied.

- **Mimer JDBC Driver Guide**

  is intended for Java application developers working with Mimer SQL. It covers all available Mimer JDBC drivers. The guide describes the usage of SQL in Java applications.

Suggestions for Further Reading

We can recommend the many works of C. J. Date. His insight into the potential and limitations of SQL, coupled with his pedagogical talents, make his books invaluable sources of study material in the field of SQL theory and usage. In particular, we can mention:

A Guide to the SQL Standard (Fourth Edition, 1997). ISBN: 0-201-96426-0. This work contains much constructive criticism and discussion of the SQL standard, including SQL-99.

SQL: 1999 - Understanding Relational Language Concepts, by Jim Melton, Alan R. Simon, and Jim Gray. ISBN: 1-55860-456-1. Explains SQL-99.

Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features, by Jim Melton. ISBN: 1-55860-677-7. In-depth guide to SQL-99's practical application.

JDBC

JDBC information can be found on the internet at:
https://www.oracle.com/technetwork/java/index.html.

For information on specific JDBC methods, please see the documentation, which is normally included in the Java development environment.

JDBC™ API Tutorial and Reference, 2nd edition. ISBN: 0-201-43328-1. A useful book published by JavaSoft.

ODBC

Microsoft ODBC 3.0 Programmer's Reference and SDK Guide for Microsoft Windows and Windows NT. ISBN: 1-57231-516-4. This manual contains information about the Microsoft Open Database Connectivity (ODBC) interface, including a complete API reference.

SQL Standards

Official documentation of the accepted SQL standards may be found in:

ISO/IEC 9075:2016(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL-2016.

CAE Specification, Data Management: Structured Query Language (SQL), Version 2. X/Open document number: C449. ISBN: 1-85912-151-9. This document contains the X/Open-95 SQL specification.

Definitions, Terms and Trademarks

ANSI	American National Standards Institute, Inc.
API	Application Programming Interface
BSQL	The Mimer SQL facility for using SQL interactively or by running a command file
ESQL	The preprocessor for embedded Mimer SQL
IEC	International Electrotechnical Commission
ISO	International Standards Organization
JDBC	The Java database API specified by Oracle Corporation
MSQL	Mimer Module SQL preprocessor.
ODBC	Open Database Connectivity
PSM	Persistent Stored Modules, the term used by ISO/ANSI for stored procedures
SQL	Structured Query Language
X/Open	X/Open is a trademark of Open Group

All other trademarks are the property of their respective holders.

Chapter 2

Mimer SQL and the ODBC API

ODBC enables your Mimer SQL database to become an integral part of an application. SQL statements can be incorporated into the application, allowing the application to retrieve and update values from a database. Values from the database can be placed in program variables for manipulation by the application. Conversely, values in program variables can be written to the database.

This chapter is not intended to be a complete guide to the functionality provided by ODBC. It is written to introduce you to accessing Mimer SQL through ODBC.

For a more detailed discussion of Mimer SQL, ODBC and other database API's, please see https://developer.mimer.com/doc/database-apis/.

The Examples in this Chapter

The ODBC function calls in the examples use ODBC 3.5 syntax, although they are not generally dependent on a Windows platform. It should be possible to use the examples as a basis for translation into other languages.

In the examples, there are various references to macros (e.g. `SQL_ERROR`), these are defined in the `sql.h` header file. `SQL_NTS` indicates a null terminated string; hopefully other names are self-explanatory.

The Mimer ODBC Driver

Mimer SQL-specific versions of ODBC functions are implemented in the Mimer SQL native ODBC driver. The driver passes SQL statements to the Mimer SQL server and returns the results of the statements to the application.

The Mimer SQL setup process automatically installs the ODBC driver when the Mimer SQL client is installed on a Windows, Linux or macOS platform.

Mimer SQL programs that are written using the ODBC API, communicate with Mimer SQL through C function calls.

Required Files

To make ODBC function calls, a C or C++ program must include the `sqlext.h` header file. By including this header file, `sqltypes.h`, `sqlucode.h` and `sql.h` are automatically included.

ODBC applications are linked with the ODBC Driver Manager. On Windows, this is the file `ODBC32.LIB`, on other platforms you will need to check the documentation supplied with the ODBC Driver Manager.

ODBC applications can also be linked directly to the native Mimer ODBC driver library, bypassing the ODBC Driver Manager.

To compile ODBC applications using Mimer specific functions and attributes, include the file `mimodbc.h`, which is supplied with the distribution. See *Mimer Specific Descriptor Fields on page 9* for details.

Unicode and ANSI Interfaces

The Mimer ODBC driver is Unicode based. This allows applications to use both the ANSI and Unicode interfaces when using Mimer SQL. Unicode based applications can both store and retrieve Unicode data through SQL statements and/or Unicode host variables.

ANSI applications can use Unicode host variables, but are restricted to character (8-bit) characters when passing data with SQL statements. Please note that database objects still have to be named with the same character set as before.

The Unicode SQL data types in Mimer SQL are called `NATIONAL CHARACTER` (or `NCHAR`), `NATIONAL CHARACTER VARYING` (`NCHAR VARYING`) and `NATIONAL CHARACTER LARGE OBJECT` (`NCLOB`).

You can find more information about these data types in the *Mimer SQL Reference Manual*. The ODBC documentation contains specifics about the `SQL_WCHAR` database type and the `SQL_C_WCHAR` and `SQL_C_WLONGVARCHAR` host language types.

External Character Set Support

The system follows the current locale setting on the machine to determine what characters are stored/retrieved when an application passes single-byte character strings to ODBC.

When character data is stored in Mimer SQL it can be stored in `CHAR` or `VARCHAR` columns or in `NCHAR` or `NVARCHAR` columns. Data in `CHAR` and `VARCHAR` columns use the Latin-1 character representation (also called ISO 8859-1). This character set can only be used to store 256 different characters. For the exact characters that can be stored see *Mimer SQL Reference Manual, Appendix B, Character Sets*. To store any other characters the data type `NCHAR` or `NVARCHAR` must be used. These column types can store **any** character.

If a locale is used by the application that has characters that are not included in Latin-1, it means that the columns in the database data must use an `NCHAR` or `NVARCHAR` column to store the correct characters. Previously, each character in the application was simply stored in a character column. When these characters were retrieved with, for example, DbVisualizer or other Unicode enabled applications, the interpretation of the characters were done differently and the wrong characters were displayed. With locale support the Mimer SQL client understands the representation of the characters in the application and maps them accordingly to its internal representation.

When retrieving data from the database, the translation works the other way. I.e. when retrieving data from a `CHAR` or `NCHAR` column to a `SQL_C_CHAR` variable, the current locale must be able to represent all the characters returned from the database. When this is not possible, a conversion error -10401 is returned. If characters stored in the database have no representation in the chosen locale, a wide character data type must be used by the application instead (`SQLWCHAR` rather than `SQLCHAR`).

Win: On Windows the setting used for the external character set is set in the Regional and Language Options in the Control Panel under the tab Advanced. This setting is used automatically by the Mimer ODBC client.

On Windows the environment variable is set to the desired code page, i.e. only numeric values may be specified (for example: 1250: ANSI Central Europe, 1251: ANSI Cyrillic, 1252: Latin1, 1253: ANSI Greek, 1254: ANSI Turkish, and so on.)

VMS: On VMS the system continues to use the Latin-1 character representation regardless of locale settings.

Linux: On other platforms (Linux, macOS, others) the application must call the runtime library routine `setlocale` to pick the locale to use. For example, the call `setlocale(LC_CTYPE, "")` sets the default locale as decided by the environment setting. The actual conversions made by the Mimer client are through the library routines `mbstowcs` (multibyte character set to wide char set) and `wcstombs`. Please note that if an application does not call `setlocale` a default 7-bit locale is used. This means that no 8-bit characters can be used without getting a conversion error. For applications where the source is not available it is possible to set an environment variable `MIMER_LOCALE` that will be used when calling the Mimer client. The value of the environment variable is used as the second argument to setlocale. For details, see the man-page for setlocale.

To use the default locale set `MIMER_LOCALE` to `current`.

The fact that the character type is considered a multi-byte character set allows any external character representation to be used. In particular various character sets such as Traditional Chinese Big5 and Japanese Shift-JIS may be used. The character set can also, of course, be a single byte character set as such as the Greek Latin-7 character set (code page 1253 on Windows). On Linux platforms the prevalent representation is UTF-8 that allows any Unicode character to be stored in a character variable.

Mimer Specific Descriptor Fields

Mimer SQL supports large objects of up to 8 terabytes. This poses a problem to ODBC applications since the ODBC API specifies length fields to be 32 bits. An ODBC compliant application is therefore unable to work with any objects larger than 2 gigabytes. The SQL/CLI standard has the same problem, so we won't get much help there.

To remedy this situation, the Mimer ODBC driver has four vendor specific descriptor attributes. Each attribute mimics the behavior of an existing attribute, the only difference is that the existing attribute is working with a 32-bit integer while ours use 64-bit integers.

C-definitions for these attributes are available in the `mimodbc.h` header file. Include this file along with the regular ODBC include files (`sql.h`, `sqlext.h`) to gain access to these features.

Note that, although these attributes are most useful when working with large objects, they may be used for any type of data.

The Mimer specific descriptor attributes:

Record field name	Type	R/W	Default
SQL_DESC_DISPLAY_SIZE_64	SQLBIGINT	ARD: Unused APD: Unused IRD: R IPD: Unused	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_LENGTH_64	SQLUBIGINT	ARD: Unused APD: Unused IRD: R IPD: R/W	ARD: Unused APD: Unused IRD: D IPD: ND
SQL_DESC_OCTET_LENGTH_64	SQLBIGINT	ARD: Unused APD: Unused IRD: R IPD: R/W	ARD: Unused APD: Unused IRD: D IPD: ND
SQL_DESC_OCTET_LENGTH_PTR_64	SQLBIGINT*	ARD: R/W APD: R/W IRD: Unused IPD: Unused	ARD: Null ptr APD: Null ptr IRD: Unused IPD: Unused

SQL_DESC_DISPLAY_SIZE_64 [IRDs]

This read-only SQLBIGINT record field contains the maximum number of characters required to display the data from the column. The value in this field is not the same as the descriptor field SQL_DESC_LENGTH because the SQL_DESC_LENGTH field is undefined for all numeric types.

SQL_DESC_LENGTH_64 [Implementation descriptors]

This SQLUBIGINT record field is either the maximum or actual character length of a character string or a binary data type. It is the maximum character length for a fixed-length data type, or the actual character length for a variable-length data type. Its value always excludes the null-termination character that ends the character string. For values whose type is SQL_TYPE_DATE, SQL_TYPE_TIME, SQL_TYPE_TIMESTAMP, or one of the SQL interval data types, this field has the length in characters of the character string representation of the datetime or interval value. Note that this field is a count of characters, not a count of bytes.

The value in this field may be different from the value for “length” as defined in ODBC 2.x. For more information, see *Microsoft ODBC 3.0 Programmer’s Reference*, Appendix D, “Data Types.”

SQL_DESC_OCTET_LENGTH_64 [Implementation descriptors]

This SQLBIGINT record field contains the length, in bytes, of a character string or binary data type. For fixed-length character or binary types, this is the actual length in bytes. For variable-length character or binary types, this is the maximum length in bytes. This value always excludes space for the null-termination character.

SQL_DESC_OCTET_LENGTH_PTR_64 [Application descriptors]

This `SQLBIGINT*` record field points to a variable that will contain the total length in bytes of a dynamic argument (for parameter descriptors) or of a bound column value (for row descriptors).

For an APD, this value is ignored for all arguments except character string and binary; if this field points to `SQL_NTS`, the dynamic argument must be null-terminated. To indicate that a bound parameter will be a data-at-execution parameter, an application sets this field in the appropriate record of the APD to a variable that, at execute time, will contain the value `SQL_DATA_AT_EXEC` or the result of the `SQL_LEN_DATA_AT_EXEC` macro. If there is more than one such field, `SQL_DESC_DATA_PTR` can be set to a value uniquely identifying the parameter to help the application determine which parameter is being requested.

If the `OCTET_LENGTH_PTR_64` field of an ARD is a null pointer, the driver does not return length information for the column. When setting the `SQL_DESC_OCTET_LENGTH_PTR_64` field to anything other than a null pointer, this field overrides the `SQL_DESC_OCTET_LENGTH_PTR` field. When both `SQL_DESC_OCTET_LENGTH_PTR` and `SQL_DESC_OCTET_LENGTH_PTR_64` are set to null pointers the driver assumes that character strings and binary values are null-terminated. (Binary values should not be null-terminated, but should be given a length to avoid truncation.)

If the call to `SQLFetch` or `SQLFetchScroll` that fills in the buffer pointed to by this field did not return `SQL_SUCCESS` or `SQL_SUCCESS_WITH_INFO`, the buffer contents are undefined. This field is a deferred field: It is not used at the time it is set, but is used at a later time by the driver to determine or indicate the octet length of the data.

Operating Systems

Mimer SQL supports ODBC as one of its native APIs for applications written in C, C++, Microsoft Visual Basic and a large number of other development tools.

Linux: Information on the availability and use of Driver Managers for ODBC on Linux and macOS platforms can be provided by your Mimer SQL distributor.

The library to specify as the ODBC driver when defining a Mimer ODBC data source is `libmimer` or `libmimodbc`. An ODBC SDK (various versions available) is also useful in order to develop applications.

As an example on how to link an ODBC application, see the supplied `ex_makefile` example makefile.

VMS: An ODBC driver is supplied with Mimer SQL on OpenVMS and client applications can link with it to use ODBC on OpenVMS platforms. The Driver Manager for ODBC on OpenVMS is not yet available. For further assistance, contact your Mimer SQL representative.

A Mimer SQL database server running on an OpenVMS node can always be accessed remotely by a client application using ODBC from another type of platform via the client/server interface.

Win: When a Mimer SQL client is installed on a Windows platform, the ODBC driver manager and other resources needed to use ODBC are also installed.

The ODBC SDK (available from Microsoft) is also required in order to develop applications.

Declarations

A C program that calls the ODBC API typically requires the following declarations:

```
#if defined(WIN32)
#include <windows.h>
#endif
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "sqlext.h"
#include "mimodbc.h"

SQLHENV  henv;       // Environment handle for application
SQLHDBC  hdbc;       // Connection handle
SQLHSTMT hstmt;      // Statement handle
```

Handles identify a particular item; in ODBC this item can be an environment, connection, statement or descriptor. When the application calls `SQLAllocHandle`, the Driver Manager creates a new item of the specified type and returns the handle to the application. The application uses the handle to identify that item when calling ODBC functions.

Initializing the ODBC Environment

The first task for any ODBC application is to initialize the ODBC environment by allocating an environment handle (`SQL_HANDLE_ENV`):

```
/* Allocate environment handle */
if ( SQLAllocHandle( SQL_HANDLE_ENV,
                     SQL_NULL_HANDLE,
                     &henv ) == SQL_ERROR )
{
    printf( "Failed to allocate environment handle\n" );
    . . .
}
```

Before an application allocates a connection, it should declare the version of ODBC that it has been written for (this mainly affects `SQLSTATE` values and datetime data types) and then allocate a connection handle:

```
/* Set the ODBC version environment */
SQLSetEnvAttr( henv,
               SQL_ATTR_ODBC_VERSION,
               (SQLPOINTER)SQL_OV_ODBC3,
               SQL_IS_INTEGER );

/* Allocate connection handle */
if ( SQLAllocHandle( SQL_HANDLE_DBC,
                     henv,
                     &hdbc ) == SQL_ERROR )
{
    printf( "Failed to allocate connection handle\n" );
    . . .
}
```

Making a Connection

If an ODBC data source has been defined, ODBC applications can connect to Mimer SQL by using the data source name. Alternatively `SQLDriverConnect` can be used.

There are a number of mechanisms to get the information required to make a connection; some applications supply the connection details, others use the ODBC dialog box to allow the user to complete the information.

The simplest form of connection uses `SQLConnect`, which requires a data source name, user ID and password, for example:

```
SQLRETURN retcode;
. . .

/* Set connection timeout - 10 seconds */
SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)10, 0);

/* Connect - DSN, User ID, Password */
retcode = SQLConnect(hdbc,
                     (SQLCHAR*) "EXAMPLEDB", SQL_NTS,
                     (SQLCHAR*) "MIMER_ADM", SQL_NTS,
                     (SQLCHAR*) "admin",
                     SQL_NTS);
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
{
    /* User connected */
```

`SQLDriverConnect` allows the driver to connect by supplying the connection information as a number of keyword-value pairs:

```
"DSN=EXAMPLEDB;UID=MIMER_ADM;PWD=admin;"
```

There is an option for the Driver Manager to enter into a dialog with the user to complete any missing connection information (the handle of the parent window needs to be supplied to use this facility).

In the following Windows example, the Driver Manager displays a window containing a combo box listing all the Mimer SQL database names and prompts for the username and password:

```
SQLCHAR     OutConnectString[256];
SQLSMALLINT StringLength;
SQLHWND     hwnd;

hwnd = GetDesktopWindow();
. . .

retcode = SQLDriverConnect( hdbc,
                            hwnd,
                            (SQLCHAR*) "DRIVER=Mimer;", SQL_NTS,
                            (SQLCHAR*)OutConnectString,
                            sizeof(OutConnectString),
                            &StringLength,
                            SQL_DRIVER_COMPLETE );
if (SQL_SUCCEEDED(retcode))
{
    /* User connected */
    printf( "connection string used: %s\n, OutConnectString );
```

Note: The macro `SQL_SUCCEEDED` replaces the test against `SQL_SUCCESS` or `SQL_SUCCESS_WITH_INFO`.

On other platforms, such as Linux, the driver does not implement a GUI popup box. Instead the user will be prompted for the required login attributes. In this case, referring to the previous example, the hwnd variable is set to null:

```
hwnd = NULL;
```

Controlling Interaction with the User

You may wish to have more control over the interaction with the user, `SQLDataSources` provides a mechanism to get information about the data sources configured on the client:

```
SQLCHAR   DSNname[SQL_MAX_DSN_LENGTH+1];
SQLCHAR   driver[33];

. . .

/* Enumerate the system data source names */
retcode = SQLDataSources( henv,
                          SQL_FETCH_FIRST_SYSTEM,
                         (SQLCHAR*)DSNname,
                          sizeof(DSNname),
                          NULL,
                          SQLCHAR*)driver,
                          sizeof(driver),
                          NULL );

while (SQL_SUCCEEDED(retcode))
{
    printf( "%-32s     %s\n", DSNname, driver );  // Display details

    /* Fetch next */
    retcode = SQLDataSources( henv,
                              SQL_FETCH_NEXT,
                             (SQLCHAR*)DSNname,
                              sizeof(DSNname),
                              NULL,
                              (SQLCHAR*)driver,
                              sizeof(driver),
                              NULL );
}
```

Connecting Using a File Data Source

Another way of making a connection is to create a file data source. The file contains keyword-value pairs to make the connection. On Windows, this file has a `.dsn` extension.

Although it is possible to include the password, this would make the system insecure and therefore is not recommended:

```
[ODBC]
DSN=EXAMPLEDB
UID=MIMER_ADM
```

To make a connection using a file data source, use the option for the Driver Manager to enter into a dialog with the user to complete any missing connection information (again, the handle of the parent window needs to be supplied to use this facility):

```
retcode = SQLDriverConnect( hdbc,
                            hwnd,
                            (SQLCHAR*) "FILEDSN=example.dsn;",
                            SQL_NTS,
                            (SQLCHAR*)OutConnectString,
                            sizeof(OutConnectString),
                            &StringLength,
                            SQL_DRIVER_COMPLETE );
if (SQL_SUCCEEDED(retcode))
{
    /* User connected */
```

Mimer Specific Keywords to SQLDriverConnect

To allow an application to connect without specifying a data source in the connection string, the following driver-specific keywords have been added for the Mimer ODBC Driver:

- PROTOCOL
- NODE
- SERVICE
- INTERFACE

The PROTOCOL keyword is mandatory for this option to be used. The regular keyword DATABASE must also be specified. Other driver-specific keywords should be used depending on the specified protocol. When PROTOCOL is specified, no data source lookup is done in the registry (Windows) or MIMER_SQLHOSTS (Linux and VMS).

Supported protocols are LOCAL (shared memory), TCP, NAMEDPIPES (only for Windows), RAPI (only for Windows), and DECNET (only for VMS).

The protocol TCP requires keyword NODE specifying the network node name. If keyword SERVICE is not specified, 1360 is used as default.

Win: The protocol NAMEDPIPES requires keyword NODE. If keyword SERVICE is not specified, the database name is used as default.

VMS: The protocol DECNET requires keyword NODE.

Note: SQLDriverConnect has a parameter that enables prompting for missing information. When PROTOCOL is specified, this is not possible.

Examples of connection strings that can be used:

```
"DRIVER={Mimer};DATABASE=cartoons;UID=mickey;PWD=mouse;PROTOCOL=local"

"DRIVER={Mimer};DATABASE=musix;UID=discux;PWD=records;PROTOCOL=local"

"DRIVER={Mimer};DATABASE=strips;UID=winnie;PWD=thepooh;PROTOCOL=tcp;
NODE=milne;SERVICE=1360"

"DRIVER={Mimer};DATABASE=pip;UID=mickey;PWD=mouse;PROTOCOL=NamedPipes;
NODE=winpix;SERVICE=pip"

"DRIVER={Mimer};DATABASE=disney;UID=donald;PWD=duck;PROTOCOL=decnet;
NODE=pictvms;INTERFACE=BG"
```

Determining Driver and Data Source Capabilities

After connection to the database, use `SQLGetInfo` to determine the capabilities of the driver and the data source associated with the connection:

```
/* Display DBMS version details */
SQLGetInfo( hdbc,
            SQL_DBMS_VER,
            (SQLPOINTER)&str_value,
            sizeof(str_value),
            &str_len );
printf( "%s\n", str_value );

/* Display SQL conformance level */
SQLGetInfo( hdbc,
            SQL_SQL_CONFORMANCE,
            (SQLPOINTER)&int_value,
            sizeof(int_value),
            NULL );
if (int_value & SQL_SC_SQL92_ENTRY)
   printf( "Entry level SQL-92\n" );
if (int_value & SQL_SC_FIPS127_2_TRANSITIONAL)
   printf( "FIPS 127-2 transitional level\n" );
if (int_value & SQL_SC_SQL92_INTERMEDIATE)
   printf( "Intermediate level SQL-92\n" );
if (int_value & SQL_SC_SQL92_FULL)
   printf( "Full level SQL-92\n" );´</pre>
```

Connecting on Linux and similar platforms

On Linux, it is possible to link an ODBC application directly to the Mimer ODBC library, `libmimodbc.so`. But, usually an ODBC Driver Manager is used, mainly to be able to handle several ODBC Data Sources. In that case the Driver Manager library is linked to the application and the Mimer ODBC library is pointed out as the Driver in the ODBC Data Source definition.

When a connection attempt is made using ODBC, a DSN (Data Source Name) is specified via one of the connection methods describe above. The ODBC Driver Manager looks up the given ODBC Data Source in an odbc.ini file. There can be a system wide `odbc.ini` file located in a known location for the platform, usually in `/etc`. Alternatively, the user can have a personal `.odbc.ini` located in the home directory. Or, the `ODBCINI` environment variable can be set to point out the data source definition file to be used.

A possible match between the given DSN and an entry in the `odbc.ini` file gives the connection information needed to load the relevant ODBC Driver dynamically and to proceed with the database access.

The following is an example of an odbc.ini file, describing two DSN specifications, with their names within straight brackets:

```
[dsn_dbcust]
Driver=/opt/MimerSQL-11.0.1A/lib/libmimodbc.so
Database=customers
Host=kixie
Port=1360

[dsn_dbext]
Driver=/opt/MimerSQL-11.0.1A/lib/libmimodbc.so
Database=external
```

In the first case above the information defined is enough to do a direct access to the database named 'customers' on the network node 'kixie', using the port number 1360. When reaching the database the user will have to provide a database username (ident name) and a password.

In the second definition there is not enough information to do a direct access. Instead the given database is looked for in the Mimer SQL database registry file called `/etc/sqlhosts`, and if found there, that information will be used to proceed with the connection. In this case, if the DSN name is the same as the database name, the Database attribute is optional.

The following are valid DSN attributes in the `odbc.ini` file when read by the Mimer ODBC Driver:

`Database`	Mimer SQL database name
`Driver`	Mimer ODBC driver library path. Or, a driver name that should be defined in the `odbcinst.ini` file, usually the name is 'mimersql'.
`Host`, `Node`, `Server` or `Servername`	Network node that the database resides on
`Port` or `Service`	TCP/IP port number
`User`, `Username` or `Uid`	Database username (ident name)
`Password` or `Pwd`	Password string (not recommended to provide this way)

Disconnecting

When the application has finished using a data source, it calls `SQLDisconnect`.

After disconnecting, the application should call `SQLFreeHandle` to release the connection handle and, if appropriate, to release the environment handle.

Error Handling

ODBC returns diagnostic information in two ways:

- a return code indicating the success or failure of the ODBC function
- diagnostics records, providing detailed information.

In general, program logic uses the return code to detect a failure and then the diagnostic records to detail the reason for the failure.

Retrieving Warning and Error Messages

If the ODBC driver returns a code indicating anything other than `SQL_SUCCESS` then the application can call `SQLGetDiagRec` to retrieve any warning or error messages:

```
SQLCHAR     msg[SQL_MAX_MESSAGE_LENGTH+1];
SQLCHAR     sqlstatus[6];
SQLSMALLINT msglen, msgno;
SQLINTEGER  nativeerror;

. . .

msgno = 1;
while (SQLGetDiagRec( SQL_HANDLE_DBC,
                      hdbc,
                      msgno++,
                      sqlstatus,
                      &nativeerror,
                      msg,
                      sizeof(msg),
                      &msglen) == SQL_SUCCESS)
{
    msg[msglen] = '\0';

    printf( "SQLSTATE:   %s\n", sqlstatus );
    printf( "Native:     %d\n", nativeerror );
    printf( "Message:    %s\n", msg );
    printf( "\n" );
}
```

Diagnostic records are associated with the ODBC handles: environment, connection, statement and descriptor. `SQLGetDiagRec` requires the handle type and the handle, making the coding of a general-purpose error handler more complex than other programming interfaces.

A warning is indicated by an `SQLSTATE` class value of '`01`' (e.g. '`01000`').

See *Appendix B Return Codes* for details.

Transaction Processing

A transaction is an essential part of database programming. It defines the beginning and end of a series of database operations that are regarded as a single unit.

For example, to transfer money between two bank accounts, an amount is subtracted from one account and the same amount is added to the other account. It is essential that either both of these operations succeed or neither does.

Mimer SQL uses a method for transaction management called Optimistic Concurrency Control (OCC). OCC does not involve any locking of rows as such, and therefore cannot cause a deadlock.

Transactions in ODBC are usually managed at the connection level, although there is the option of applying a commit or rollback for all connections within an environment.

Transaction Management Mode

There are two modes for managing transactions within ODBC, Autocommit and Manual-commit. `SQLSetConnectAttr` is used to switch between the modes.

Autocommit Mode

Autocommit mode is the default transaction mode for ODBC; when a connection is made, it is in autocommit mode until `SQLSetConnectAttr` is used to switch to manual commit mode.

In autocommit mode each individual statement is automatically committed when it completes successfully, no explicit transaction management functions are necessary.

However, the return code from the function must still be checked as it is possible for the implicit transaction to fail.

Manual-commit Mode

When in manual commit mode, all executed statements are included in the same transaction until calling `SQLEndTran` specifically completes it.

When an application turns autocommit off, the next statement against the database starts a transaction. The transaction continues until `SQLEndTran` is called with either `SQL_COMMIT` or `SQL_ROLLBACK`. The next command sent to the database after that starts a new transaction.

Completing Transactions

Transactions are completed (either committed or rolled back) by use of the ODBC function `SQLEndTran` rather than using the `SQL COMMIT` or `ROLLBACK` statements.

Calling `SQLEndTran` with a request to rollback a transaction causes Mimer SQL to discard any changes made since the start of the transaction and to end the transaction.

Example Transaction

```
/* Disable transaction autocommit mode */
SQLSetConnectAttr( hdbc, SQL_ATTR_AUTOCOMMIT,
                   (SQLPOINTER)SQL_AUTOCOMMIT_OFF, 0 );

SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt );

retry:
/* First statement against Mimer SQL starts a transaction */
SQLExecDirect( hstmt,
               "UPDATE mimer_store.currencies \
                   SET exchange_rate = exchange_rate * 1.05 \
                   WHERE code = 'USD'", SQL_NTS );

SQLExecDirect( hstmt,
               "UPDATE mimer_store.currencies \
                   SET exchange_rate = exchange_rate * 1.08 \
                   WHERE code = 'GBP'", SQL_NTS );

printf( "Commit transaction? : " );
scanf( "%s", ans );
if (ans[0] == 'Y'
|| ans[0] == 'y')
{
   retcode = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT );
   if (retcode == SQL_ERROR)
   {
      /* Check SQLSTATE for transaction conflict */
      SQLGetDiagField (hdbc,
                       1,
                       SQL_DIAG_SQLSTATE,
                       sqlstatus,
                       sizeof(sqlstatus),
                       &msglen );
      if (strcmp( sqlstatus, "40001" ) == 0) goto retry;
      goto display_error;
   }
}
else
{
   SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_ROLLBACK );
}
```

Setting the Transaction Isolation Level

To set the transaction isolation level, use the `SQL_ATTR_TXN_ISOLATION` connection attribute.

The default isolation level for Mimer SQL is `SQL_TXN_REPEATABLE_READ`.

Executing a Command

The simplest way to execute a statement is to execute it directly using the `SQLExecDirect` function.

Each `INSERT`, `UPDATE` and `DELETE` statement returns the number of rows affected by the operation, the function `SQLRowCount` returns this count.

For example:

```
SQLINTEGER rowcount;
. . .

/* Allocate statement handle */
SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt );
SQLExecDirect( hstmt,
               "UPDATE mimer_store.currencies \
                   SET exchange_rate = exchange_rate * 1.05 \
                   WHERE code = 'USD'", SQL_NTS );

SQLRowCount( hstmt, &rowcount );
printf( "%d rows have been updated\n", rowcount );
```

Repeating – Prepared Execution

Where an SQL statement will be repeatedly executed, it is more usual to use prepared execution, as a means to reduce the parsing and compilation overheads.

Mimer SQL reduce the performance difference between direct and prepared execution by maintaining and re-using compiled statements on the server.

Prepared Statement Example

In this example each of the parameters in the prepared SQL statement (indicated by `?`) are bound to a variable in the application before the statement is executed:

```
SQLFLOAT   increase;
SQLCHAR    code[4];
SQLINTEGER increaseInd, codeInd;
. . .

SQLPrepare( hstmt,
            "UPDATE mimer_store.currencies \
                SET exchange_rate = exchange_rate * ? \
                WHERE code = ?", SQL_NTS );

SQLBindParameter( hstmt, 1, SQL_PARAM_INPUT, SQL_C_DOUBLE,
                  SQL_REAL, 7, 0,
                  &increase, 0, &increaseInd );
SQLBindParameter( hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                  SQL_CHAR, 4, 0,
                  code, sizeof(code), &codeInd );

/* Set parameter values and length/indicator */
increase = 1.05;
strcpy( code, "USD" );
codeInd = SQL_NTS;

SQLExecute( hstmt );
SQLRowCount( hstmt, &rowcount );
printf( "%d rows have been updated\n", rowcount );
```

Stored Procedure Example

Similarly, it is possible to prepare an SQL statement that calls a stored procedure:

```
SQLINTEGER  order_id, item_id;
SQLSMALLINT quantity;
SQLINTEGER  orderInd = 0, itemInd = 0, quantityInd = 0;
. . .

SQLPrepare( hstmt,
            "{CALL mimer_store.order_item( ?, ?, ? )}",
            SQL_NTS );

SQLBindParameter( hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
                  SQL_INTEGER, 0, 0,
                  &order_id, SQL_IS_INTEGER, &orderInd );
SQLBindParameter( hstmt, 2, SQL_PARAM_INPUT, SQL_C_SLONG,
                  SQL_INTEGER, 0, 0,
                  &item_id, SQL_IS_INTEGER, &itemInd );
SQLBindParameter( hstmt, 3, SQL_PARAM_INPUT, SQL_C_SSHORT,
                  SQL_INTEGER, 0, 0,
                  &quantity, SQL_IS_SMALLINT, &quantityInd );

/* Set parameter values */
order_id = 700001;
item_id = 60158;
quantity = 2;

SQLExecute( hstmt );
```

Parameters in Procedure Calls

Parameters in procedure calls can be input, input/output, or output. A more complicated example illustrates how to handle an output parameter:

```
SQLCHAR     country[3];
SQL_INTERVAL_STRUCT interval;
SQLINTEGER  countryInd, intervalInd;
SQLSMALLINT numparams;
. . .

SQLPrepare( hstmt,
            "{CALL mimer_store.age_of_adult( ?, ? )}",
            SQL_NTS );

SQLBindParameter( hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
                  SQL_CHAR, 3, 0,
                  country, sizeof(country), &countryInd );
SQLBindParameter( hstmt, 2, SQL_PARAM_OUTPUT, SQL_C_INTERVAL_YEAR,
                  SQL_INTERVAL_YEAR, 0, 0,
                  &interval, sizeof(SQL_INTERVAL_STRUCT),
                  &intervalInd );

SQLNumParams( hstmt, &numparams );
printf( "statement contains %d parameters\n", numparams );

/* Set input parameter value and length */
strcpy( country, "US" );
countryInd = SQL_NTS;

SQLExecute( hstmt );
printf( "%d years\n", interval.intval.year_month.year );
```

A statement handle is released by calling `SQLFreeHandle`; however, it is more efficient to reuse statement handles rather than freeing them and allocate new ones. When `SQLFreeHandle` is called, the driver releases the associated structure. `SQLDisconnect` automatically frees all statements on a connection.

Result Set Processing

There are two ways of processing a result set. One method uses `SQLBindCol` to bind applications variables to the columns of the result set. The second method of processing the result set is to use `SQLGetData`.

Using SQLBindCOL

When each row of data is fetched, the column data is copied to the application variables. The following example also illustrates how to use the indicator variable; this either returns the length of character data (a negative length indicates that truncation has taken place), or `SQL_NULL_DATA` if the data is null:

```
SQLCHAR    code[4];
SQLCHAR    currency[33];
SQLINTEGER codeInd, currencyInd;
. . .
/* Allocate statement handle */
SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt );

SQLExecDirect( hstmt,
               "SELECT code, currency \
                   FROM mimer_store.currencies",
               SQL_NTS );

SQLBindCol( hstmt, 1, SQL_C_CHAR,
            code, sizeof(code), &codeInd );
SQLBindCol( hstmt, 2, SQL_C_CHAR,
            currency, sizeof(currency), &currencyInd );

while ((retcode = SQLFetch( hstmt )) != SQL_NO_DATA)
{
   printf( "%s  %s\n", code, currency );
}

/* Close the cursor */
SQLCloseCursor( hstmt );
```

Using SQLGetData

The second method of processing the result set is to use `SQLGetData`; the equivalent of the previous example can be written:

```
SQLExecDirect( hstmt,
               "SELECT code, currency \
                   FROM mimer_store.currencies",
               SQL_NTS );

while ((retcode = SQLFetch( hstmt )) != SQL_NO_DATA)
{
   SQLGetData( hstmt, 1, SQL_C_CHAR,
               code, sizeof(code), &codeInd );
   SQLGetData( hstmt, 2, SQL_C_CHAR,
               currency, sizeof(currency), &currencyInd );

   printf( "%s  %s\n", code, currency );
```

Combining Result Set Processing Methods

SQLBindCol and SQLGetData can be combined. The previous two examples used forward-only cursors, which means that they only support fetching rows serially from the start to the end of the cursor.

In modern screen-based application, the user expects to be able to scroll backwards and forwards through the data. While it is possible to cache small result sets in memory on the client, this is not feasible when dealing with large result sets. Scrollable cursors provide the answer.

Scrollable Cursors

Scrollable cursors allow you to move forward and backward to any row within the result set. A statement attribute of SQL_SCROLLABLE specifies that the cursor will be opened in scroll mode.

The function SQLFetchScroll supports fetching the next, prior, first and last rows, as well as absolute and relative positioning.

For example:

```
/* Allocate statement handle */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

/* Set cursor scrollable */
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_SCROLLABLE,
                         (SQLPOINTER)SQL_SCROLLABLE, 0);
if (retcode == SQL_ERROR) goto error;

SQLExecDirect(hstmt,
              "SELECT code, currency \
               FROM mimer_store.currencies \
               WHERE code LIKE 'A%'",
               SQL_NTS);

SQLBindCol(hstmt, 1, SQL_C_CHAR,
           code, sizeof(code), &codeInd);
SQLBindCol(hstmt, 2, SQL_C_CHAR,
           currency, sizeof(currency), &currencyInd);

printf("Original sort order\n");
while ((SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0)) != SQL_NO_DATA)
   printf("%s  %s\n", code, currency);

printf("\nReverse order\n");
while ((SQLFetchScroll(hstmt, SQL_FETCH_RELATIVE, -1)) != SQL_NO_DATA)
   printf("%s  %s\n", code, currency);

/* Close the cursor */
SQLCloseCursor(hstmt);
```

Updating Data

Applications can update data by executing the UPDATE, DELETE and INSERT statements.

An alternative method is to position the cursor on a particular row and then use DELETE CURRENT, or UPDATE CURRENT statements.

The following example illustrates how this can be done by using two statement handles:

```
SQLHSTMT   cscroll, cupdate;
SQLCHAR    code[4];
SQLCHAR    currency[33];
SQLINTEGER codeInd, currencyInd;
. . .

/* Allocate statement handles */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &cscroll);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &cupdate);

/* Set scroll cursor attributes */
SQLSetStmtAttr(cscroll, SQL_ATTR_CURSOR_SCROLLABLE,
               (SQLPOINTER)SQL_SCROLLABLE, 0);
SQLSetStmtAttr(cscroll, SQL_ATTR_CONCURRENCY,
               (SQLPOINTER)SQL_CONCUR_VALUES, 0);

/* Name the cursor */
SQLSetCursorName(cscroll, "CRN", SQL_NTS);

SQLExecDirect(cscroll,
              "SELECT code, currency \
               FROM mimer_store.currencies \
               FOR UPDATE OF currency",
               SQL_NTS);

SQLBindCol(cscroll, 1, SQL_C_CHAR,
           code, sizeof(code), &codeInd);
SQLBindCol(cscroll, 2, SQL_C_CHAR,
           currency, sizeof(currency), &currencyInd);

/* Set the update cursor to use optimistic concurrency */
SQLSetStmtAttr(cupdate, SQL_ATTR_CONCURRENCY,
               (SQLPOINTER)SQL_CONCUR_VALUES, 0);

/* Prepare the positioned update statement using scroll cursor name */
SQLPrepare(cupdate,
           "UPDATE mimer_store.currencies \
               SET currency = ? \
            WHERE CURRENT OF crn",
            SQL_NTS);

/* Bind the code parameter in the update statement */
SQLBindParameter(cupdate, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
                 SQL_CHAR, 33, 0,
                 currency, sizeof(currency), &currencyInd);

/* Position within the result set on the scrolling cursor */
SQLFetchScroll(cscroll, SQL_FETCH_ABSOLUTE, 3);
SQLSetPos(cscroll, 1, SQL_POSITION, SQL_LOCK_NO_CHANGE);

/* Update currency name using update statement handle */
if (strncmp(currency, "Leke", 4) == 0)
   strcpy(currency, "Albanian Leke");
else
   strcpy(currency, "Leke");
currencyInd = SQL_NTS;

SQLExecute(cupdate);
```

Native SQL Escape Clauses

Using native SQL escape clauses with short-form syntax is supported by the Mimer SQL Experience server, not only from the ODBC driver, but from all Mimer database API's, including scripted or interactive execution from BSQL.

Long-form escape clauses are supported, but need to be handled by the ODBC driver before execution.

Short- and long-form syntax examples:

```
{ d'2001-01-01' }
--(* vendor(Microsoft), product(ODBC) d'2001-01-01' *)--
```

The function `SQLNativeSQL` transforms a long-form escape clause to a short-form escape clause, making it directly executable.

```
SQLNativeSql(hdbc,
    "UPDATE mimer_store.items SET release_date =
    --(* vendor(Microsoft), product(ODBC) d '2001-01-01' *)--
    WHERE product_id = 100",
    SQL_NTS, (keep)szSqlStr, 1200, pcbSqlStr);
SQLExecDirect(hstmt, szSqlStr, SQL_NTS);
```

The only accepted vendor is Microsoft, and the only accepted product is ODBC. All long-form escape clauses with other vendors or products will be discarded. A syntactically incorrect long-form escape clause will not be transformed.

Escaped functions

The first column lists functions supported in escaped function calls.

The second column tells if a function is supported in an unescaped form as well.

The third column describes the regular Mimer SQL equivalence, when different from the escaped function call form.

Escaped function	Also un-escaped	Mimer SQL equivalence
`{fn ABS(value)}`	yes	
`{fn ACOS(value)}`	yes	
`{fn ASCII(string)}`	no	`ASCII_CODE(string)`
`{fn ASIN(value)}`	yes	
`{fn ATAN(value)}`	yes	
`{fn ATAN2(value1,value2)}`	yes	
`{fn BIT_LENGTH(string)}`	yes	
`{fn CEILING(value)}`	yes	
`{fn CHAR(string)}`	no	`ASCII_CHAR(string)`
`{fn CHAR_LENGTH(string)}`	yes	
`{fn CHARACTER_LENGTH(string)}`	yes	

Escaped function	Also un-escaped	Mimer SQL equivalence
{fn CONCAT(string1,string2, ...)}	yes	string1 || string2 ...
{fn COS(value)}	yes	
{fn COT(value)}	yes	
{fn CURDATE()}	no	CURRENT_DATE
{fn CURRENT_DATE()}	no	CURRENT_DATE
{fn CURRENT_TIME()}	no	-
{fn CURRENT_TIMESTAMP()	no	-
{fn CURTIME()}	no	-
{fn DATABASE()}	no	-
{fn DAYNAME(value)}	no	ODBC.DAYNAME(value)
{fn DAYOFMONTH(value)}	yes	EXTRACT(DAY FROM value)
{fn DAYOFWEEK(value)}	yes	
{fn DAYOFYEAR(value)}	yes	
{fn DEGREES(value)}	yes	
{fn DIFFERENCE(string,string)}	no	-
{fn EXP(value)}	yes	
{fn EXTRACT(field FROM value)}	yes	
{fn FLOOR(value)}	yes	
{fn HOUR(value)}	yes	EXTRACT(HOUR FROM value)
{fn IFNULL(expr1,expr2,...)}	yes	COALESCE(expr1,expr2,...)
{fn INSERT(str1,start,len,str2)}	no	PASTE(str1,start,len,str2)
{fn LCASE(string)}	no	LOWER(string)
{fn LEFT(string, length)}	yes	SUBSTRING(string FROM 1 FOR length)
{fn LENGTH(string)}	no	CHAR_LENGTH(TRIM(TRAILING FROM string))
{fn LOCATE(str1,str2[,start])}	yes	
{fn LOG(value)}	no	LN(value)
{fn LOG10(value)}	yes	
{fn LTRIM(string)}	no	TRIM(LEADING FROM string)
{fn MINUTE(value)}	yes	EXTRACT(MINUTE FROM value)
{fn MOD(value,value)}	yes	

Escaped function	Also un-escaped	Mimer SQL equivalence
{fn MONTH(value)}	yes	EXTRACT(MONTH FROM value)
{fn MONTHNAME(value)}	no	ODBC.MONTHNAME(value)
{fn NOW()}	no	LOCALTIMESTAMP
{fn OCTET_LENGTH(string)}	yes	
{fn PI()}	no	-
{fn POSITION(substring, source)};	yes	POSITION(substring IN source)
{fn POWER(value, value)}	yes	
{fn QUARTER(value)}	yes	
{fn RADIANS(value)}	yes	
{fn RAND()}	no	CAST(IRAND() AS DOUBLE PRECISION) / 2147483647
{fn REPEAT(string,value)}	yes	
{fn REPLACE(source,str1,str2)}	yes	
{fn RIGHT(string,value)}	yes	
{fn ROUND(value,number)}	yes	
{fn RTRIM(string)}	no	TRIM(TRAILING FROM string)
{fn SECOND(value)}	yes	EXTRACT(SECOND FROM value)
{fn SIGN(value)}	yes	
{fn SIN(value)}	yes	
{fn SOUNDEX(value)}	yes	
{fn SPACE(value)}	no	REPEAT(' ', value)
{fn SQRT(value)}	yes	
{fn SUBSTRING(string,start[,length])	yes	SUBSTRING(string FROM start [FOR length])
{fn TAN(value)}	yes	
{fn TIMESTAMPADD(tsi_type,val1,val2)}	no	-
{fn TIMESTAMPDIFF(tsi_type,val1,val2)}	no	-
{fn TRUNCATE(value,digits)}	yes	
{fn UCASE(string)}	no	UPPER(string)
{fn USER()}	no	CURRENT_USER
{fn WEEK(value)}	yes	

Escaped function	Also un-escaped	Mimer SQL equivalence
`{fn YEAR(value)}`	yes	`EXTRACT(YEAR FROM value)`

Chapter 3

Mimer SQL and the JDBC API

JDBC is the de-facto standard for accessing relational database systems from the Java programming language. It defines a framework that provides a uniform interface to a number of different database connectivity modules.

Mimer SQL supports JDBC as one of its native application programming interfaces (API).

The Mimer JDBC Driver is a Type 4 - Native Protocol All-Java Driver, also known as a Java thin driver. The Type 4 architecture uses a message protocol that is specific to Mimer SQL; as this means that there is no need for any intervening processes or translation, this architecture is extremely efficient.

This chapter is not intended to be a complete guide to the functionality provided by JDBC. It is written to introduce you to accessing Mimer SQL through JDBC.

The Mimer JDBC Driver

The Mimer JDBC Driver is required in order to access a Mimer SQL database over TCP/IP from a Java application.

You can use the Mimer JDBC driver on any client computer that has the Java runtime environment is installed.

The driver is extremely small in size, so that it can be simply incorporated into an applet that can be downloaded over the Internet.

The Mimer JDBC driver is distributed with the Mimer SQL software. You can also download it from: https://developer.mimer.com/downloads. No other Mimer SQL software is required to be installed on the Java client, eliminating any need for configuration management on the client side.

To install Java, download the Java SDK. You can find the latest version here: https://www.oracle.com/technetwork/java/index.html

All further information about the Mimer JDBC drivers, including programming aspects, are found in the *Mimer JDBC Driver Guide*.

Chapter 4

Embedded SQL

In previous chapters, we discussed the ODBC and JDBC APIs. This chapter discusses the scope, principles, processing and structure of embedded SQL (ESQL).

ESQL enables you to code SQL statements in a host program written in C/C++, COBOL or Fortran. You can specify SQL statements directly in the host program's source code. However, because the host language's compiler won't recognize the SQL statements as valid, a preprocessor is required.

The Scope of Embedded Mimer SQL

The following groups of SQL statements are common to ESQL and interactive SQL:

- Data manipulation statements for reading or changing the contents of the database and invoking stored routines. These are basically similar between interactive SQL and ESQL, but differ in certain details as a result of the different environments in which the statements are used.
- Transaction control statements for grouping database operations in transactions (indivisible units of work).
- Access control statements for allocating privileges and access rights to users of the system. These are identical between interactive SQL and ESQL.
- Data definition statements for creating and altering objects in the database. These are identical between interactive SQL and ESQL.
- Connection statements for identifying the current user of the system.
- System administration statements for controlling the availability of the database and its physical components, managing backups and updating database statistics.

There are a number of commands provided for use with BSQL which are not included in the Mimer SQL interface, these are described in the *Mimer SQL User's Manual, Chapter 9, Mimer BSQL*.

Note: In the *Mimer SQL Reference Manual, Chapter 12, Usage Modes*, Mimer SQL statements are identified as valid for use in ESQL, for interactive use or both.

General Principles for Embedding SQL Statements

The following sections discuss host languages, preprocessors, identifying SQL statements, code, comments and recommendations.

Host Languages

You can embed Mimer SQL statements in application programs written in C/C++, COBOL or Fortran. The basic principles for writing ESQL programs are the same in all languages and all ESQL statements are embedded in the same way.

Information given in this manual applies to all languages unless otherwise explicitly stated. Language-specific information is detailed in *Host Language Dependent Aspects* on page *307*.

ESQL Preprocessors

Because host language compilers do not recognize ESQL statements as valid, an ESQL preprocessor is required. An ESQL preprocessor processes the SQL statements embedded in a host language.

Linux: Mimer SQL supports an ESQL preprocessor for the C/C++ and Fortran host languages on Linux platforms.

VMS: Mimer SQL supports an ESQL preprocessor for the C/C++, COBOL and Fortran host languages on OpenVMS platforms.

Win: Mimer SQL supports an ESQL preprocessor for the C/C++ host language on Windows platforms.

Identifying SQL Statements

SQL statements are included in the host language source code exactly as though they were ordinary host language statements (i.e. they follow the same rules of conditional execution, etc., which apply to the host language).

SQL statements are identified by the leading keywords `EXEC SQL` (in all host languages) and are terminated by a language-specific delimiter. Every separate SQL statement must be delimited in this way.

Blocks of several statements may not be written together within one set of delimiters. For instance, in COBOL, two consecutive `DELETE` statements must be written as:

```
EXEC SQL DELETE FROM countries END-EXEC.
EXEC SQL DELETE FROM producers END-EXEC.
```

and not

```
EXEC SQL DELETE FROM countries
    DELETE FROM producers END-EXEC.
```

Single SQL statements can however be split over several lines, following the host language rules for line continuation.

The following embedded statement is thus acceptable in a Fortran program (the continuation mark is a + in column 6 on the second line):

```
 EXEC SQL DELETE FROM countries
+    WHERE code = 'BA' END-EXEC.
```

The keywords `EXEC SQL` may not be split over more than one line.

Included Code

Any code which is included in the program by the host language compiler (as directed by host language `INCLUDE` statements) is not recognized by the ESQL preprocessor.

If external source code modules containing SQL statements are to be included in the program, the non-standard `SQL INCLUDE` statement must be used, for example:

```
EXEC SQL INCLUDE 'filename'
```

Files included in this way are physically integrated into the output from the preprocessor.

Comments

Comments may be written in the ESQL program according to the rules for writing comments in the host language. Thus comments may be written within an SQL statement if the host language accepts comments within host language statements.

The following statement is valid in C/C++:

```
exec sql DELETE FROM countries  /* Remove Bosnia and Herzegovina */
             WHERE code = 'BA';
```

Note: The keywords `EXEC` and `SQL` may not be separated by a comment.

Recommendations

We recommend the following, when using ESQL:

- Avoid variable names beginning with the letters `SQL` (except for `SQLSTATE` and `SQLCODE`, which should be used when appropriate).
- Avoid subroutine or subprogram names ending with a number.

Language-specific restrictions are described in *Host Language Dependent Aspects* on page *307*.

Processing ESQL

The following sections discuss preprocessing and processing ESQL.

Preprocessing – the ESQL Command

An application program containing ESQL statements must first be preprocessed using the `ESQL` command before it can be passed through the host language compiler, since the host language itself does not recognize the ESQL syntax.

Preprocessors are available for the host languages supported on each platform, see *Host Languages* on page *34*.

The input to the preprocessor is thus a source code file containing host language statements and ESQL statements.

The output from the preprocessor is a source code file in the same host language, with the ESQL statements converted to source code data assignment statements and subroutine calls that pass the SQL statements to the Mimer SQL database manager.

The original ESQL statements are retained as comments in the output file, to help in understanding the program if a source code debugger is used.

The output from the preprocessor is human-readable source code, still retaining a large part of the structure and layout of the original program, which is used as input to the appropriate host language compiler to produce object code.

The default file extensions for preprocessor input and output files depend on the host language used and are shown in the table below:

Language	Input file extension	Output file extension
C	`.ec`	`.c`
C header	`.eh`	`.h`
COBOL	`.eco`	`.cob`
Fortran	`.efo`	`.for`

Invoking the ESQL Preprocessor

The ESQL preprocessor has the following syntax:

```
esql [-c|-h|-b|-f] [-l] [-n] infile [outfile]

esql [--c|--header|--cobol|--fortran] [--line] [--nologo] infile [outfile]

esql [-v|--version] | [-?|--help]
```

Language

Unix-style	VMS-style	Function
`-c` `--c`	`/C`	Indicates that the input file is written using the C/C++ host language.
`-h` `--header`	`/HEADER`	Indicates that the input is a C/C++ host language header file.
`-b` `--cobol`	`/COBOL`	Indicates that the input file is written using the COBOL host language.
`-f` `--fortran`	`/FORTRAN`	Indicates that the input file is written using the Fortran host language.

Options

Unix-style	VMS-style	Function
`-?` `--help`	`/HELP`	Display usage information.

Unix-style	VMS-style	Function
`-l` `--line`	`/LINE`	Generates #line preprocessing directives for source written in the C language. These force the C compiler to produce diagnostic messages with line numbers relating to the input C source code rather than the code generated by the preprocessor (and thus compiled by the C compiler.)
`-n` `--nologo`	`/NOLOGO`	Suppresses the display of the copyright message and input filename on the screen (warnings and errors are always displayed on the screen.)
`-v` `--version`	`/VERSION`	Display version information.

Input-file and Output-file

Unix-style	VMS-style	Function
`infile`	`infile`	The input-file containing the source code to be preprocessed. If no file extension is specified, the appropriate file extension for the source language is assumed (previously described in this section.)
`[outfile]`	`[outfile]`	The output-file which will contain the compiler source code generated by the preprocessor. If not specified, the output file will have the same name as the input file, but with the appropriate default output file extension (previously described in this section.)

Note: As an application programmer, you should never attempt to directly modify the output from the preprocessor.

Any changes that may be required in a program should be introduced into the original ESQL source code. Mimer Information Technology AB cannot accept any responsibility for the consequences of modifications to the preprocessed code.

File Format Handling

When the ESQL preprocessor reads the input file it needs to make decisions about what kind of file format that is used. If the input file has a leading BOM (Byte Order Mark), which is especially common on the Windows platform, the file format is assumed to be according to this information. For example, this can indicate that the file is UTF-8 or UTF-16.

If no BOM is located, which usually is the case on other platforms than Windows, the file format is presumed to be in line with the current locale setting.

If the input file is written in plain ASCII, without using a BOM, the file format is not an issue. All steps in the build process for the source file will likely work without problems. In other cases, if a specific encoding is used, the file format must be considered in the translation and execution environments used.

The output file produced by ESQL is of the same format as the input file.

Example

The following example, on OpenVMS, shows how to preprocess the `DSQLSAMP` program:

```
$ ESQL/C MIMER$EXAMPLES:DSQL
```

What Does the Preprocessor Do?

The preprocessor checks the syntax and to some extent the semantics of the ESQL statements. (See *Handling Errors and Exceptions* on page *69* for a more detailed discussion of how errors are handled). Syntactically invalid statements cannot be preprocessed and the source code must be corrected.

Processing ESQL – the Compiler

The output from the ESQL preprocessor is compiled in the usual way using the appropriate host language compiler, and linked with the appropriate routine libraries.

Linux: On Linux platforms, the gcc and gfortran compilers are supported.

VMS: The following compilers are supported on the OpenVMS platform:

- DEC C, VSI C
- DEC Fortran, VSI Fortran
- DEC COBOL, VSI COBOL

Note: For COBOL, the source program must be formatted according to the ANSI rules. Use the `/ANSI` option when compiling the resulting COBOL program.

Win: On Windows platforms, the C compiler identified by the `cc` symbol in the file `.\dev\samples\makefile.mak` below the installation directory is supported.

Note: Other compilers, from other software distributors, may or may not be able to compile the ESQL preprocessor output. Mimer Information Technology cannot guarantee the result of using a compiler that is not supported.

The SQL Compiler

At run-time, database management requests are passed to the SQL compiler responsible for implementing the SQL functions in the application program.

The SQL compiler performs two functions:

- It checks SQL statements semantically against the data dictionary.
- It optimizes operations performed against the database (i.e. internal routines determine the most efficient way to execute the SQL request, with regard to the existence of secondary indexes and the number of rows in the tables addressed by the statement). You, as a programmer, do not need to worry, for instance, about the order in which tables are addressed in a complex selection condition. This optimization process is completely transparent.

Note: Since all SQL statements are compiled at run-time, there can be no conflict between the state of the database at the times of compilation and execution. Moreover, the execution of SQL statements is always optimized with reference to the current state of the database.

Essential Program Structure

All application programs using embedded Mimer SQL must include certain basic components, summarized below in the order in which they appear in a program.

1 Host Variable Declarations

A host variable is a variable used in the embedded program for entering data to the database or retrieving data from the database. Host variables must be declared inside the `SQL DECLARE SECTION` to be recognized. Host variables can be used in embedded statements where an expression can be used.

See the section *Using Host Variables* on page *46* for more details.

2 The Status Information Variable: SQLSTATE

The status information variable `SQLSTATE`, if used, must be declared inside the `SQL DECLARE SECTION`. This variable provides the application with status information for the most recently executed SQL statement.

3 Executable SQL Statements

This is the body of the program, and performs the required operations on the database. Normally, these begin with connecting to Mimer SQL and performing the required transactions before finally disconnecting from Mimer SQL.

Summary of Functions for Manipulating Data

The following table summarizes the functions for data manipulation in interactive SQL and ESQL.

Operation	Interactive SQL	ESQL
Retrieve data	`SELECT` generates a result table directly.	Declare a cursor for the `SELECT` statement. The cursor must be opened and positioned. Data is retrieved into host variables one row at a time with `FETCH`. Alternative: `SELECT INTO` retrieves a single-row result set directly into host variables.
Update data	`UPDATE` operates on a set of rows or columns.	`UPDATE` operates on a set of rows. `UPDATE CURRENT` operates on a single row through a cursor.
Insert data	`INSERT` inserts one or many rows at a time.	`INSERT` inserts one or many rows at a time.
Delete data	`DELETE` operates on a set of rows.	`DELETE` operates on a set of rows. `DELETE CURRENT` operates on a single row through a cursor.

Operation	Interactive SQL	ESQL
Invoke routine	`CALL` is used to execute all stored procedures, i.e. both result set and non-result set procedures are handled the same way. Functions can be specified where an expression could be used and are invoked when an expression used in the same context would be evaluated.	Result set procedures are called by using the `CALL` clause in a cursor declaration and then using `FETCH`. The `CALL` statement is used directly for non-result set procedures. Functions can be specified where an expression could be used and are invoked when an expression used in the same context would be evaluated.
Assignment	`SET`	The set statement can be used to assign values to a host variable. E.g. if you want to invoke a user defined function or method and assign the result to a host variable, a statement such as `SET :hv = Capitalize('john brown')`, can be used.

Many SQL statements (e.g. data definition statements) are simply embedded in their logical place in the application program and are executed without direct reference to other parts of the program.

Some features of ESQL however require special consideration, and are dealt with in detail in the chapters that follow:

- Access authorization through the use of user and program idents.
- Data manipulation statements which require the use of cursors (`FETCH`, `UPDATE CURRENT`, `DELETE CURRENT`). These together with cursor handling statements are probably the most commonly used statements in ESQL.
- Transaction control, which is essential for a consistent database.
- Dynamic SQL, which is a special set of statements allowing an application program to process SQL statements entered by the user at run-time.
- Exception handling, which controls the action taken when, for instance, the end of a result set is reached.

Linking Applications

Linux: The example makefile `ex_makefile`, found in the installation examples directory, provides a verified example of the recommended way to build C applications on Linux platforms.

For Fortran, see the `ex_makefile_f` example makefile.

Applications built using the procedure contained in this makefile will reference the Mimer SQL shared library called `libmimer`.

VMS: All Mimer SQL applications should be linked with the options file `MIMER$SQL.OPT`, as shown in the following example:

```
$ LINK main,MIMER$LIB:MIMER$SQL/OPT
```

The `MIMER$SQL.OPT` file includes the following:

- `MIMER$LIB:MIMER$SQL.EXE` (shareable library)

If an image linked in this fashion is activated, it will translate the logical name `MIMER$SQL` to get the name of the Mimer SQL shareable library to be used.

The logical name is defined by the `SYS$MANAGER:MIMER$SETUP_xxxxx` command procedure.

Win: The example makefile `.\dev\samples\makefile.mak` in the installation directory should be copied and used in the recommended way to build applications on Windows platforms.

If applications are linked as recommended above to reference the Mimer SQL shared library, they will automatically use a new version of Mimer SQL when it is installed, without having to be re-linked.

Connecting to a Database

A database in Mimer SQL refers to the complete collection of databanks that may be accessed from one Mimer SQL system.

Mimer ESQL supports the ability to change between different connections (i.e. access different databases) from within the same application program. An application program may have several database connections open simultaneously, although only one is active at any one time.

Only idents of type USER are allowed to log on to Mimer SQL.

The CONNECT Statement

Logging on is requested from an application program with the CONNECT statement, see the *Mimer SQL Reference Manual, Chapter 12, CONNECT*, for the syntax description.

The CONNECT statement establishes a connection between a USER ident and a database

```
exec sql CONNECT TO 'db' AS 'con1' USER 'ident' USING 'pswd';
```

To connect using an OS_USER login with the same name as the current operating system user, provide an empty ident name string. E.g.

```
exec sql CONNECT TO 'db' AS 'con2' USER ' ' USING ' ';
```

Local and Remote Databases

A connection may be established to any local or remote database, which has been made accessible from the current machine, see the *Mimer SQL System Management Handbook, Chapter 3, Creating a Mimer SQL Database*, for details, by specifying the database by name or by using the keyword DEFAULT.

Default or Named Database

If the keyword DEFAULT is used, an OS_USER login is used for the connection attempt.

```
exec sql CONNECT TO DEFAULT;
```

If the database name is given as an empty string, the DEFAULT database is used.

```
exec sql CONNECT TO ' ' AS 'con1' USER 'ident' USING 'pswd';
```

The database may be given an explicit connection name for use in DISCONNECT and SET CONNECTION statements. If no explicit name is given, the database name is used as the connection name.

```
exec sql CONNECT TO 'db' USER 'ident' USING 'pswd';
```

Implicit Connection

Normally, CONNECT should be the first SQL statement executed in an application program using ESQL. However, if another SQL statement is issued before any connection has been established in the current application, an implicit connection will be attempted.

An implicit connection is made to the DEFAULT database using the current operating system user.

In order for the implicit connect attempt to be successful, the current operating system user must be defined as an OS_USER login in Mimer SQL and the DEFAULT database must be defined as a local database on the machine on which the current operating system user is defined.

If an implicit connection has previously been established in the application and there is no current connection, issuing an executable statement will result in a new attempt to make the same implicit connection. However, if an explicit connection has previously been established in the application and there is no current connection, issuing an executable statement will cause an error.

Changing Connection

A connection established by a successful CONNECT statement is automatically active.

An application program may make multiple connections to the same or different databases using the same or different idents, provided that each connection is identified by a unique connection name.

Only the most recent connection is active. Other connections are dormant, and may be made active by the SET CONNECTION statement. Resources such as cursors used by a connection are saved when the connection becomes dormant, and are restored by the appropriate SET CONNECTION statement.

The statement sequence below connects to a user-specific database as a specified ident name and to the DEFAULT database using an OS_USER login. The user-specific connection is initially active. Then the DEFAULT connection is activated. Finally the user-specific connection is activated again using SET CONNECTION.

```
EXEC SQL CONNECT TO 'db' AS 'con1' USER 'ident' USING 'pswd';
...
EXEC SQL CONNECT TO DEFAULT;
...
-- Set activate connection to CON1
EXEC SQL SET CONNECTION 'con1';
```

Note: If different connections are made with different idents, the apparent access rights of the application program may change when the current connection is changed.

Disconnecting

The DISCONNECT statement breaks the connection between a user and a database and frees all resources allocated to that user for the specified connection (all cursors are closed and all compiled statements are dropped). The connection to be broken is specified as the connection name or as one of the keywords ALL, CURRENT or DEFAULT. (If a transaction is active when the DISCONNECT is executed, an error is raised and the connection remains open).

A connection does not have to be active in order to be disconnected. If an inactive connection is broken, the application still has uninterrupted access to the database through the current (active) connection, but the broken connection is no longer available for activation with SET CONNECTION.

If the active connection is broken, the application program cannot access the database until a new CONNECT or SET CONNECTION statement is issued.

Note: The distinction between breaking a connection with DISCONNECT and making a connection inactive by issuing a CONNECT or SET CONNECTION for a different connection is, a broken connection has no saved resources and cannot be reactivated by SET CONNECTION.

The table below summarizes the effect on the connection con1 of CONNECT, DISCONNECT and SET CONNECTION statements depending on the state of the connection.

Statement	con1 non-existent	con1 current	con1 inactive
CONNECT TO db1 AS con1	con1 current	error – connection already exists	error – connection already exists
DISCONNECT con1	error – connection does not exist	con1 disconnected	con1 disconnected
SET CONNECTION con1	error – connection does not exist	ignored	con1 made current
CONNECT TO db2 AS con2	–	con1 made inactive	con1 unaffected
DISCONNECT con2	–	con1 unaffected	con1 unaffected
SET CONNECTION con2	–	con1 made inactive	con1 unaffected

PROGRAM Idents – ENTER and LEAVE

PROGRAM idents may be entered from within an application program by using the ENTER statement, see the *Mimer SQL Reference Manual, Chapter 12, ENTER* for the syntax description. This statement must be issued in a context where a user is already connected as PROGRAM idents cannot connect directly to the system.

When a PROGRAM ident is entered, any privileges granted to that ident become current and privileges belonging to the previous ident (i.e. the ident issuing the ENTER statement) are suspended. However, any cursors opened by the previous ident remain open.

PROGRAM idents are disconnected with the LEAVE statement. If LEAVE is requested with the optional keyword RETAIN, the full environment of the PROGRAM ident being left is kept.

Cursors left open by the PROGRAM ident are deactivated but not closed, and retain their positions in the respective result tables. The environment is restored if the PROGRAM ident is re-entered.

If LEAVE is requested without RETAIN, the environment of the PROGRAM ident being left is dropped. This means that all cursors and compiled statements are destroyed.

Note: The distinction between leaving a `PROGRAM` ident with the option `RETAIN` and entering a new `PROGRAM` ident is, while both operations save the environment of the `PROGRAM` ident, cursors left open at `ENTER` may still be used but those left open at `LEAVE RETAIN` are inaccessible until the program ident is re-entered.

The statements `ENTER` and `LEAVE` may not be issued within transactions, see *Transaction Handling and Database Security* on page *221*.

Communicating with the Application Program

Information is transferred between the embedded SQL (ESQL) application program and the Mimer SQL database manager in four ways:

- through host variables used in SQL statements
- through the status variable SQLSTATE
- through the diagnostics area, accessed by the SQL statement GET DIAGNOSTICS
- through an SQL descriptor area.

Using Host Variables

Host variables are used in SQL statements to pass values between the database and the application program.

Declaring Host Variables

All variables used in SQL statements must be declared for the preprocessor. This is done by enclosing the variable declarations between the SQL statements BEGIN DECLARE SECTION and END DECLARE SECTION.

The following example in C declares the character variables user and passw for use in SQL statements:

```
int rc, pf, cnt;

exec sql BEGIN DECLARE SECTION;
   char user[129],
        passw[129]; /* 128 character column, and a null byte */
exec sql END DECLARE SECTION;
```

Any variables declared outside the DECLARE SECTION will not be recognized by the preprocessor.

Variables are declared within the section using the normal host language syntax.

Variables which are not used in SQL statements may also be declared in the SQL DECLARE SECTION. (This will however extend the symbol table established by the preprocessor more than is necessary.)

The use of array variables is currently not supported in embedded Mimer SQL (except for character string variables).

Using Variables in Statements

Host variables may be used:

- to receive information from the database (SELECT INTO, FETCH, CALL and SET statements)
- to assign values to columns in the database (CALL, INSERT and UPDATE statements)
- to manipulate information taken from the database or contained in other variables (in expressions)
- to get descriptor and diagnostics information (GET DESCRIPTOR, SET DESCRIPTOR and GET DIAGNOSTICS)
- in dynamic SQL statements.

In all these contexts, the data type of the host variable or database column must be compatible with the data type of the corresponding database value or host variable. General considerations of data type compatibility may be found in the *Mimer SQL Reference Manual*. Host language specific aspects are described in *Host Language Dependent Aspects* on page *307* of this manual.

If you have an `INTEGER` column containing values that do not fit into the largest integer variable allowed on your machine (remember that Mimer SQL supports `INTEGER` values with a precision of up to 45 digits), you can, for example, use a character string or float host variable for that column. In this case, Mimer SQL automatically performs the necessary conversions.

Host variables are preceded by a colon when used in SQL statements, see the *Mimer SQL Reference Manual, Chapter 6, Host Identifiers*.

Note: The colon is not part of the host variable, and should not be used when the variable is referenced in host language statements.

Example

```
EXEC SQL SELECT COUNT(*)
            INTO :VAR
            FROM table
            WHERE condition;

if VAR < LIMIT then ...
```

Indicator Variables

In ESQL, indicator variables associated with main variables are used to handle null values in database tables.

Indicator variables should be an exact numeric data type with scale zero and are declared in the same way as main variables in the `SQL DECLARE SECTION`.

See *Declarations* on page *309* for a description of how main and indicator variables should be declared in the specific host languages.

Indicator variables are used in SQL statements by either specifying the name of the indicator variable, preceded by a colon, after the main variable name or by using the keyword `INDICATOR`, for example:

```
:main_variable :indicator_variable
```

or

```
:main_variable INDICATOR :indicator_variable
```

Transfer from Tables to Host Variables

When a null value is retrieved into a host variable by a `FETCH`, `SELECT INTO`, `EXECUTE`, `SET` or `CALL` statement, the value of the main variable is undefined and the value of the indicator variable is set to `-1`.

An error occurs if the main variable is not associated with an indicator variable in the SQL statement. It is therefore recommended as a precaution that indicator variables are used for all columns which are not defined as `NOT NULL` in the database.

An indicator variable should always be used when a host variable is used for a routine parameter with mode `OUT` or `INOUT` because a null value can always be returned via a routine parameter.

When a non-null value is assigned to a main variable associated with an indicator variable, the indicator variable is set to zero or a positive value. A positive value indicates that the value assigned to a main character variable was truncated, and gives the length of the original value before truncation.

Transfer from Host Variables to Tables

When the host variable associated with an indicator variable is used to assign a value to a column, the value assigned is null if the value of the indicator variable is set to `-1`.

In such a case, the value of the main variable is irrelevant. If the indicator variable has a value of zero or a positive value, or if the main variable is not associated with an indicator variable, the value of the main variable itself is assigned to the column.

External Character Set Support

The handling of the single byte character data types follows the current locale setting on the machine to determine what characters are stored/retrieved when an embedded SQL application passes single-byte character strings to the Mimer client.

When character data is stored in Mimer SQL it can be stored in `CHAR`, `VARCHAR` or `CLOB` columns, or in `NCHAR`, `NVARCHAR` or `NCLOB` columns. Data in `CHAR`, `VARCHAR` and `CLOB` columns use the Latin-1 character representation (also called ISO 8859-1). This character set can only be used to store 256 different characters. For the exact characters that can be stored see *Mimer SQL Reference Manual, Appendix B, Character Sets*. To store any other characters the data type `NCHAR`, `NVARCHAR` or `NCLOB` must be used. These column types can store **any** character.

If a locale is used by the application that has characters that are not included in Latin-1, it means that the columns in the database data must use an `NCHAR`, `NVARCHAR`, or `NCLOB` column to store the correct characters. With the locale support the Mimer SQL client understands the representation of the characters in the application and maps them accordingly to its internal representation.

When retrieving data from the database, the translation work the other way. I.e. when retrieving data from a `CHAR` or `NCHAR` column to a single-byte character variable, the current locale must be able to represent all the characters returned from the database. When this is not possible, a conversion error -10401 is returned. If characters stored in the database have no representation in the chosen locale, a wide character data type must be used by the application instead (e.g. the C type `wchar_t` rather than `char`).

This means that applications using older versions of Mimer may have to be updated to work with the new version. Typically the data type used in the database is altered from `CHAR` to `NCHAR`, or from `VARCHAR` to `NVARCHAR`. This is done with the `ALTER TABLE` statement (see *Mimer SQL Reference Manual, Chapter 12, ALTER TABLE*). Other possible changes is to switch from a character representation (e.g. `char`) to a Unicode representation (e.g. `wchar_t`) for the application variables, or to switch to a locale that can handle all relevant characters.

On Windows the setting used for the external character set is set in the Regional and Language Options in the Control Panel under the tab Advanced. This setting is used automatically by the Mimer client.

On VMS the system continues to use the Latin-1 character representation regardless of locale settings.

On other platforms (Linux, macOS, others) the application must call the runtime library routine `setlocale` to pick the locale to use. For example, the call `setlocale(LC_CTYPE, "")` sets the default locale as decided by the environment setting. The actual conversions made by the Mimer client are through the library routines `mbstowcs` (multibyte character set to wide char set) and `wcstombs`. Please note that if an application does not call `setlocale` a default 7-bit locale is used. This means that no 8-bit characters can be used without getting a conversion error. For applications where the source is not available it is possible to set an environment variable `MIMER_LOCALE` that will be used when calling the Mimer client. The value of the environment variable is used as the second argument to `setlocale`.

To use the default locale set `MIMER_LOCALE` to `current`. On Windows the environment variable is set to the desired code page, i.e. only numeric values may be specified (for example: 1250: ANSI Central Europe, 1251: ANSI Cyrillic, 1252: Latin1, 1253: ANSI Greek, 1254: ANSI Turkish, and so on.)

The fact that the character type is considered a multi-byte character set allows any external character representation to be used. In particular various character sets such as Traditional Chinese Big5 and Japanese Shift-JIS may be used. The character set may, of course, be a single byte character set as such as the Greek Latin-7 character set (code page 1253 on Windows). On Linux platforms the prevalent representation is UTF-8 that allows any Unicode character to be stored in a character variable.

The SQLSTATE Variable

The `SQLSTATE` variable provides the application, in a standardized way, with return code information about the most recently executed SQL statement.

`SQLSTATE` must be declared between the `BEGIN DECLARE SECTION` and the `END DECLARE SECTION` (i.e. in the SQL declare section), as a 5 character long string (excluding any terminating null byte).

The return codes provided by `SQLSTATE` can contain digits and capital letters.

`SQLSTATE` consists of two fields. The first two characters of SQLSTATE indicates a class, and the following three characters indicates a subclass. Class codes are unique, but subclass codes are not. The meaning of a subclass code depends on the associated class code.

To determine the category of the result of an SQL statement, the application can test the class of `SQLSTATE` according to the following:

SQLSTATE Class	Result category
`00`	Success
`01`	Success with warning
`02`	No data
Other	Error

For a list of SQLSTATE values, see *Return Codes* on page *323*.

The Diagnostics Area

The diagnostics area holds status information for the most recently executed SQL statement.

There is always one diagnostics area for an application, no matter how many connections the application holds.

Information from the diagnostics area is selected and retrieved by the `GET DIAGNOSTICS` statement. The syntax for `GET DIAGNOSTICS` (including a description of the diagnostics area) is described in *Mimer SQL Reference Manual, Chapter 12, GET DIAGNOSTICS*.

The `GET DIAGNOSTICS` statement does not change the contents of the diagnostics area, except it does set `SQLSTATE`.

The SQL Descriptor Area

An SQL descriptor area is used to hold data and descriptive information required for execution of dynamic SQL statements. SQL descriptor areas are allocated and maintained by ESQL statements, described in the *Mimer SQL Reference Manual*.

The SQL descriptor area is discussed in detail in *SQL Descriptor Area* on page *63*.

Accessing Data

This section explains how embedded SQL applications retrieve data.

Retrieving Data Using Cursors

Data is retrieved from database tables with the `FETCH` statement, which fetches the values from an individual row in a result set into host variables.

The result set is defined by a `SELECT` construction or a result set procedure `CALL`, see *Manipulating Data* on page *261*, used in a cursor declaration. A cursor may be thought of as a pointer which moves through the rows of the result set as successive `FETCH` statements are issued.

An exception is raised to indicate when the `FETCH` has reached the end of the result set.

Data retrieval involves several steps in the application program code, which are as follows:

- declaration of host variables to hold data
- declaration of a cursor with the appropriate `SELECT` conditions or result set procedure `CALL`
- opening the cursor
- performing the `FETCH`
- closing the cursor.

General Framework

The steps in the previous section are built into the application program as shown in the general frameworks below (only SQL statements are shown in the frameworks).

For a `SELECT`:

```
EXEC SQL BEGIN DECLARE SECTION;
   ... VAR1, VAR2, ... VARn ...
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cursor-name CURSOR FOR select-statement;

EXEC SQL OPEN cursor-name;

loop as required
   EXEC SQL FETCH cursor-name
                INTO :VAR1, :VAR2, ..., :VARn;
end loop;

EXEC SQL CLOSE cursor-name;
```

For a result set procedure `CALL`:

```
EXEC SQL BEGIN DECLARE SECTION;
   ... VAR1, VAR2, ... VARn ...
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cursor-name CURSOR FOR CALL routine-invocation;

EXEC SQL OPEN cursor-name;

loop as required
    EXEC SQL FETCH cursor-name
                 INTO :VAR1, :VAR2, ..., :VARn;
end loop;

EXEC SQL CLOSE cursor-name;
```

Declaring Host Variables

All host variables used to hold data fetched from the database and used in selection conditions or as result set procedure parameters must be declared within an SQL `DECLARE SECTION`, see *Communicating with the Application Program* on page *46*.

Indicator variables for columns that may contain null values must also be declared.

The same indicator variable may be associated with different main variables at different times, but declaration of a dedicated indicator variable for each main variable is recommended for clarity.

Declaring the Cursor

A cursor operates as a row pointer associated with a result set.

A cursor is defined by the `DECLARE CURSOR` statement and the set of rows addressed by the cursor is defined by the `SELECT` statement in the cursor declaration.

Cursors are local to the program in which they are declared. A cursor is given an identifying name when it is declared.

`DECLARE CURSOR` is a declarative statement that does not result in any implicit connection to a database, see *Idents and Privileges* on page *217* for details on connecting to a database.

Preprocessing the statement generates a series of parameters used by the SQL compiler but does not generate any executable code; the select-expression or result set procedure call in the cursor declaration is not executed until the cursor is opened.

Holdable cursors can be declared using the `WITH HOLD` clause. An open cursor declared `WITH HOLD` remain open after `COMMIT`.

Cursors should normally be declared `WITHOUT HOLD` (default), because `WITH HOLD` cursors require more internal resources then ordinary cursors. In addition, long lasting `WITH HOLD` cursors can have negative performance effects just like long lasting transactions.

If the cursor declaration contains a `CALL` to a result set procedure, it is `FETCH` that actually executes the procedure.

The `RETURN` statement is used from within the result set procedure to return a row of the result set.

Each `FETCH` causes statements in the result set procedure to execute until a `RETURN` statement is executed, which will return the row data defined by it. Execution of the procedure is suspended at that point until the next `FETCH`.

If, during execution, the end of the procedure is encountered instead of a `RETURN` statement, the `FETCH` result is end-of-set. See *Result Set Procedures* on page *264* for a detailed description of result set procedures.

Note: It is advisable to always use an explicit list of items in the `SELECT` statement of the cursor declaration. The shorthand notations `SELECT *` and `SELECT table.*` are useful in interactive SQL, but can cause conflicts in the variable lists of `FETCH` statements if the table definition is changed.

Host Variables

The cursor declaration can use host variables in the `WHERE` or `HAVING` clause of the `SELECT` statement.

The result set addressed by the cursor is then determined by the values of these host variables at the time when the cursor is opened.

The same cursor declaration can thus address different result sets depending on when the cursor is opened, for example:

```
EXEC SQL DECLARE C1 CURSOR..; -- cursor with host variables
set variables
EXEC SQL OPEN C1;             -- open one result set
...
EXEC SQL CLOSE C1;
set variables
EXEC SQL OPEN C1;             -- open different result set
```

Scrollable cursors can be declared using the `SCROLL` keyword. When a cursor is declared as scrollable, records can be fetched using an orientation specification. This makes it possible to scroll through the result set with the cursor.

Cursors which are to be used only for retrieving data may be declared with a `FOR READ ONLY` clause in the `SELECT` statement. This can improve performance slightly in comparison with cursors that permit update and delete operations.

Opening the Cursor

A declared cursor must be opened with the OPEN statement before data can be retrieved from the database. The OPEN statement evaluates the cursor declaration in terms of

- the privileges the current user holds on any tables and views accessed by the cursor
- the values of any host variables used in the SELECT clause
- for a cursor calling a result set procedure, whether the current user has the required EXECUTE privilege on the procedure and also the values of any IN parameters

When the OPEN statement has been executed, the cursor is positioned before the first row in the result set.

Retrieving Data

Once a cursor has been opened, data may be retrieved from the result set with FETCH statements, see the *Mimer SQL Reference Manual, Chapter 12, FETCH*, for the syntax description.

Host variables in the variable list correspond in order to the column names specified in the SELECT clause of the cursor declaration. The number of variables in the FETCH statement may not be more than the number of columns selected. The number of variables may be less than the number of columns selected, but a 'success with warning'-code is then returned in SQLSTATE.

A suitably declared record structure may be used in place of a variable list in host languages where this is supported, see *Host Language Dependent Aspects* on page *307*.

Each FETCH statement moves the cursor to the specified row in the result set before retrieving data. In strict relational algebra, the ordering of tuples in a relation (the formal equivalent of rows in a table) is undefined. The SELECT statement in the cursor declaration may include an ORDER BY clause if the ordering of rows in the result set is important to the application.

Note: A cursor declared with an ORDER BY clause cannot be used for updating table contents.

If no ORDER BY clause is specified, the ordering of rows in the result set is unpredictable.

Note: The variables into which data is fetched are specified in the FETCH statement, not in the cursor declaration. In other words, data from different rows in the result set may be fetched into different variables.

When there are no more rows to fetch, the exception condition NOT FOUND will be raised.

The following construction thus fetches rows successively until the result set is exhausted:

```
EXEC SQL DECLARE C1 CURSOR FOR select-statement;
EXEC SQL OPEN C1;

EXEC SQL WHENEVER NOT FOUND GOTO done;
LOOP
    EXEC SQL FETCH C1 INTO :var1,:var2,...,:varn;
END LOOP

done:
EXEC SQL CLOSE C1;
```

Access Rights

The access rights for a user are checked when the cursor is opened and they remain unchanged for that cursor until the cursor is closed.

For example, if an application program declares and opens a cursor, then SELECT access on the table is revoked from the user running the program, data can still be fetched from the result set as long as the cursor remains open. Any subsequent attempt to open the same cursor will, however, fail.

Block Fetching

The Embedded SQL interface tries whenever possible to fetch rows in blocks to minimize server communications. The first fetch would normally issue a request to the server for a number of rows at once. In most situations, this will improve application performance.

In some situations, this is not the desired behavior. One such situation is queries searching through a huge number of rows without the help of indexes. For example if the database server is only able to return one row a second, and the entire query takes minutes, the user can still be happy as long as he sees the first rows on screen. If this is important to the application, set the fetch size manually. An appropriate fetch size is the number of rows displayed at once. See *Mimer SQL Reference Manual, Chapter 12, SET SESSION FETCH SIZE* for more information.

Closing a Cursor

An opened cursor remains open until it is closed with one of the statements CLOSE, COMMIT, ROLLBACK or DISCONNECT. CLOSE closes the specified cursor. ROLLBACK and DISCONNECT close all open cursors for the connection. COMMIT closes all open cursors for the connection, except cursors declared as WITH HOLD. Once a cursor is closed, the result set is no longer accessible. However, the cursor declaration remains valid, and a new cursor may be opened with the same declaration.

Note: The result set addressed by the new cursor may not be the same if the contents of the database or the values of variables used in the declaration have changed.

Normally, resources used by the cursor remain allocated when the cursor is closed and will be used again if the cursor is re-opened. The optional form CLOSE cursor-name RELEASE deallocates cursor resources. Use of CLOSE with the RELEASE option is recommended in application programs which open a large number of cursors, particularly where system resources are limited.

Note: The use of CLOSE with the RELEASE option may slow down performance if there is a following OPEN, since it requires that new resources are allocated at the next OPEN for that cursor. For this reason it should only be used when necessary.

Cursors are local to a connection and remain open but dormant when the connection is made dormant. The state of dormant cursors is fully restored (including result set addressed and position in the result set) when the connection is reactivated. Cursors are, however, closed and cursor resources are deallocated, when a connection is disconnected.

Note: Cursors opened in a program ident context are closed and resources deallocated when LEAVE is executed within the same connection, unless LEAVE RETAIN is specified.

Retrieving Single Rows

If the result of a SELECT statement is known to be a single row, the SELECT INTO statement may be used as an alternative to fetching data through a cursor.

This is a much simpler programming construction, since cursors are not used and the only requirement is that host variables used in the SELECT INTO statement are declared in the DECLARE SECTION.

However, there are two disadvantages associated with SELECT INTO:

- An error occurs if the result set addressed by the search condition contains more than one row. In other words, SELECT INTO can only be reliably used when there is no possibility of a multi-row result set (essentially when the search condition includes the columns that form a UNIQUE or PRIMARY KEY column or returns just the result of a set function, e.g. COUNT(*)).
- Execution of the SELECT INTO statement involves a check that the result set contains one single row, which may incur unnecessary overhead. Even if it is known that the result row is unique, a single FETCH operation through a cursor may be a more efficient implementation.

Use of a SELECT INTO statement is justified when the result set may contain several rows, but it is a condition for continued execution of the application program that the result row is unique. With a cursor, this would require a construction that checked that one and only one FETCH operation could be performed (alternatively, use a separate SELECT COUNT with the same search condition as the cursor). In such a case, a SELECT INTO statement with a check on the return code, see *Handling Errors and Exceptions* on page *69*, is probably the preferred solution.

A CALL statement can be used to return information to the one or more host variables associated with the output parameter(s) of the procedure.

A SET statement can be used with a function or method invocation to return information to one host variable.

Retrieving Data from Multiple Tables

Data can be retrieved from multiple tables in ESQL by addressing several tables in the SELECT statement of the cursor declaration, in the same way as in interactive SQL. The preprocessor generates a SELECT statement addressing multiple tables, which is optimized by the SQL compiler when the cursor is opened.

Example

```
EXEC SQL DECLARE c_1 CURSOR FOR SELECT ...
                                    FROM a JOIN b
                                    ON a.x = b.y;
EXEC SQL OPEN c_1;
...
```

An alternative way to link information between tables could be to define the search condition for one cursor in terms of a variable fetched through another cursor:

```
EXEC SQL DECLARE c_1 CURSOR FOR SELECT x
                                FROM a;
EXEC SQL DECLARE c_2 CURSOR FOR SELECT ...
                                FROM b
                                WHERE y = :HOSTX;

EXEC SQL OPEN c_1;
EXEC SQL FETCH c_1
           INTO :HOSTX;
EXEC SQL CLOSE c_1;

EXEC SQL OPEN c_2;
EXEC SQL FETCH c_2;
...
```

When considering the two alternatives, the first one is preferred. The reason for this is:

- The SQL optimizer gets the full information about the query that it is supposed to return a result set for. In this way the optimizer can make more use of statistical information and it can thereby optimize the query to execute in a more efficient way.
- The application will require less resources in the form of open cursors.
- If the application is run in a client/server environment, the second alternative will cause more communication over the network, since it will send data over the net which is only used to determine which data from the second cursor that will be selected and is of no real interest to the application.
- The application will be more compact as well as easier to understand and maintain.

The 'Parts explosion' Problem

A special case of data retrieval from multiple tables is the use of stacked cursors to fetch data from logical copies of the same table, in a manner that provides a solution to the so called “Parts explosion” problem.

A cursor can be defined as `REOPENABLE` and the same cursor may be opened several times in succession in the same application program, each previous instance of the cursor being saved on a stack and restored when the following instance is closed. A `FETCH` statement refers to the most recently opened instance of a cursor. Each instance of the cursor addresses an independent result set and the position of each cursor in its own result set is saved on the stack.

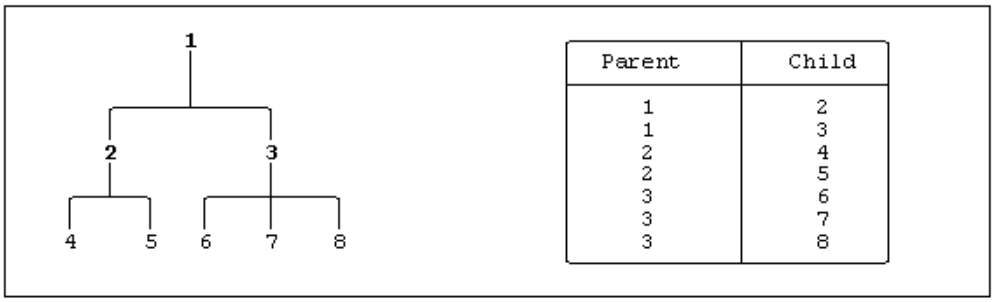
Note: Result sets addressed by different instances of a cursor may differ according to the conditions prevailing when the cursor instance was opened.

The state of the cursor stack needs to be controlled by the application. A counter can be used to indicate if there are more instances of the cursor remaining on the stack. See the example that follows.

Stacked cursors are typically used in application programs which traverse a tree structure stored in the database.

For example (this is a simplified variant of the "parts explosion" problem), traverse a tree structure and print out the leaf nodes:

Tree **Relational table**



Parent	Child
1	2
1	3
2	4
2	5
3	6
3	7
3	8

```
procedure TRAVERSE;
integer CSTACK, LASTC;
EXEC SQL BEGIN DECLARE SECTION;
   integer PARENT, CHILD;
   string SQLSTATE(5);
EXEC SQL END DECLARE SECTION;
begin
   EXEC SQL DECLARE c_tree REOPENABLE CURSOR FOR
                           SELECT parent, child
                              FROM tree
                              WHERE parent = :PARENT;

   CSTACK := 1;
   LASTC := 1;
   PARENT := 1;                       -- Start at root node

   EXEC SQL OPEN CTREE;

   loop
      EXEC SQL FETCH c_tree
                  INTO :PARENT, :CHILD;

      if SQLSTATE = "02000" then      -- No more children
         EXEC SQL CLOSE c_tree;       -- Pop the parent
         CSTACK := CSTACK - 1;
         exit when CSTACK = 0;
         if CSTACK >= LASTC then
            print(PARENT);            -- Write leaf node
         end if;
         LASTC := CSTACK;
      else                            -- Step to next level
         PARENT := CHILD;
         EXEC SQL OPEN c_tree;        -- Stack the current parent
                                         and open new level
         CSTACK := CSTACK + 1;
      end if;
   end loop;
end TRAVERSE;
```

The counters `CSTACK` and `LASTC` keep track of the number of stacked cursor levels and the latest level in the tree hierarchy respectively.

Entering Data into Tables

The following sections explain how to perform cursor-independent operations and update and delete using cursors.

Cursor-independent Operations

The SQL statements `CALL`, `INSERT`, `DELETE` and `UPDATE`, as well as user-defined functions or method invocations, embedded in application programs operate on a set of rows in a table or view in exactly the same way as in interactive SQL.

Host variables may be used in the statements to supply values or set search conditions, and host variables may be used as routine parameters.

Examples:

```
EXEC SQL INSERT INTO mimer_store.items(item_id,
                                       product_id, format_id,
                                       release_date,
                                       price, stock, reorder_level,
                                       ean_code,
                                       producer_id)
                    VALUES (CURRENT VALUE FOR mimer_store.item_id_seq,
                            :product_id, :format_id,
                            mimer_store.cast_to_date(:book_release_date),
                            :book_price, :book_stock, :book_reorder_level,
                            (:ean * 10) + mimer_store.ean_check_digit(:ean),
                            producer_id);
```

From the standpoint of the application program, each statement is a single indivisible operation, regardless of how many columns and rows are affected.

Updating and Deleting Through Cursors

The `UPDATE CURRENT` and `DELETE CURRENT` statements (see the *Mimer SQL Reference Manual, Chapter 12, SQL Statements* for the syntax description), allow update and delete operations to be controlled on a row-by-row basis from an application. These statements operate through cursors, which are declared and opened as described above for `FETCH`.

These statements operate on the current row of the cursor referenced in the statement. If there is no current row, e.g. the cursor has been opened but not yet positioned with a `FETCH` statement, an error is raised.

`UPDATE CURRENT` changes the contents of the current row according to the `SET` clause in the statement, but does not change the position of the cursor. Two consecutive `UPDATE CURRENT` statements will therefore update the same row twice.

`DELETE CURRENT` deletes the current row and does not move the cursor; after a `DELETE CURRENT` statement, the cursor is positioned between rows and there is no current row. The cursor must be moved to the next row with a `FETCH` statement before any other operation can be performed through the cursor.

For both `UPDATE CURRENT` and `DELETE CURRENT` statements, the table name as used in the statement must be exactly the same as the table name addressed in the cursor declaration. The cursor must also address an updatable result set.

`UPDATE CURRENT` and `DELETE CURRENT` changes for a particular cursor can be divided into several transactions if the cursor is a holdable cursor. A cursor declared `WITH HOLD` remains open when transactions are committed, which makes it possible to use the same cursor for fetch and update of additional rows after `COMMIT`. However, each row must still be fetched and updated (or deleted) in the same transaction.

All SELECT statements are by default read only. This means that they cannot be used with UPDATE CURRENT and DELETE CURRENT unless a FOR UPDATE clause is added to the SELECT statement. If a FOR UPDATE OF clause is used to specify which fetched columns may be updated, only the columns specified may appear in the corresponding UPDATE statement.

Cursors can not be updatable if the data retrieval statement in the cursor declaration contains any of the following features at the top level (i.e. not in a subquery) of the statement:

- reference to more than one table in the FROM clause (i.e. an explicit or implicit join)
- reference to a read-only view in the FROM clause
- the keyword DISTINCT
- set-functions in the SELECT list (AVG, COUNT, MAX, MIN, SUM)
- arithmetic or string concatenation expressions in the SELECT list
- a GROUP BY clause
- an ORDER BY clause
- the UNION keyword
- the EXCEPT keyword
- the INTERSECT keyword
- a CALL to a result set procedure
- the SELECT statement is implicitly or explicitly declared as READ ONLY. (To be updatable, the SELECT needs to be declared with the FOR UPDATE clause.)

When to Use UPDATE CURRENT, DELETE CURRENT

UPDATE CURRENT and DELETE CURRENT statements are useful for manipulating single rows in interactive applications where rows are displayed, and the user decides which rows to delete or update.

The example below illustrates the program framework for such an operation (the construction is similar for a `DELETE CURRENT` operation):

```
...
EXEC SQL DECLARE c_1 CURSOR FOR SELECT ... FOR UPDATE;
...
EXEC SQL OPEN c_1;
EXEC SQL WHENEVER NOT FOUND GOTO done;
loop
   EXEC SQL FETCH c_1
               INTO :VAR1, :VAR2, ..., :VARn;

   display VAR1, VAR2, ..., VARn;
   prompt "Update this row?";
   if ANSWER = "YES" then
      prompt "Give new values";
      EXEC SQL UPDATE tab
                  SET col1 = :NEWVAL1,
                      col2 = :NEWVAL2, ...
                  WHERE CURRENT OF c_1;
      display "Row updated";
   end if;

   prompt "Display next row?";
   exit when ANSWER = "NO";
end loop;

done:
EXEC SQL CLOSE c_1;
```

In situations where there is no requirement to interactively choose rows and where all the rows to be updated or deleted can be specified completely in terms of a `WHERE` clause, it is more efficient to do so rather than use a cursor.

An operation completely specified as a `WHERE` clause is executed as a single statement, rather than a series of statements (i.e. one for each `FETCH` etc.).

Dynamic SQL

This section discusses the principles of dynamic SQL, processing dynamic SQL, the descriptor area, preparing statements, extended dynamic cursors and prepared statements.

Principles of Dynamic SQL

Dynamic SQL enables you to execute SQL statements placed in a string variable instead of explicitly writing the statements inside a program. This allows SQL statements to be constructed within an application program. These facilities are typically used in interactive environments, where SQL statements are submitted to the application program from the terminal.

An example of when dynamic SQL is needed would be a program for interactive SQL, where any correct SQL statement may be entered at the terminal and processed by the application. Limited dynamic facilities may however be provided by relatively simple application programs.

SQL Statements and Dynamic SQL

The following classes of SQL statements may be submitted to programs using dynamic SQL. Statements excluded from dynamic applications are declarations, diagnostic statements and dynamic SQL statements themselves.

- **Access control statements:**

  ```
  ENTER
  LEAVE
  ```

- **Data definition statements:**

  ```
  CREATE
  ALTER
  COMMENT
  DROP
  ```

- **Security control statements:**

  ```
  GRANT
  REVOKE
  ```

- **Transaction control statements:**

  ```
  SET SESSION
  SET TRANSACTION
  START
  COMMIT
  ROLLBACK
  ```

- **Data manipulation statements:**

  ```
  CALL
  SELECT
  SELECT INTO
  INSERT
  UPDATE
  UPDATE CURRENT
  DELETE
  DELETE CURRENT
  COMPOUND STATEMENT
  SET
  ```

- **System administration statements:**

  ```
  CREATE BACKUP
  ALTER DATABANK RESTORE
  SET DATABASE
  SET DATABANK
  SET SHADOW
  UPDATE STATISTICS
  DELETE STATISTICS
  ```

Submitting Statements

Statements may be submitted to dynamic SQL applications in two forms:

- Fully defined statements, written exactly as they would be submitted to interactive SQL. For example:

```
GRANT SELECT ON mimer_store_book.details TO mimer_admin_group

SELECT code, country FROM mimer_store.countries
```

- Statements with parameter markers, which identify positions where the value of a host variable will be inserted when the statement is executed or the cursor is opened. A parameter marker is represented by a question mark `?` or using colon notation. For example:

```
UPDATE mimer_store.currencies
SET exchange_rate = ?
WHERE code = ?

DELETE FROM countries
WHERE code = :codeparam

SELECT currency_code
FROM mimer_store.countries
WHERE code LIKE '%' || ? || '%'
```

  Statements submitted with parameter markers are equivalent to normal embedded statements using host variables, except that the statements are defined at run-time.

General Summary of Dynamic SQL Processing

The following statements are used when SQL statements are dynamically submitted:

Statement	Description
`ALLOCATE CURSOR`	Allocate extended cursor.
`ALLOCATE DESCRIPTOR`	Allocate SQL descriptor area.
`CLOSE`	Close an open cursor.
`DEALLOCATE DESCRIPTOR`	Deallocate SQL descriptor area.
`DEALLOCATE PREPARE`	Deallocate prepared SQL statement.
`DECLARE CURSOR`	Declare a cursor for a statement which will be dynamically submitted.
`DESCRIBE`	Examine the object form of the statement and assign values to the appropriate parameters in the SQL descriptor area.
`EXECUTE`	Execute a prepared statement (except result set generating statements).
`EXECUTE IMMEDIATE`	Shorthand form for PREPARE followed by EXECUTE. This form can only be used for fully-defined non-result set statements with no parameter markers.
`FETCH`	Fetch rows for a dynamic cursor.
`GET DESCRIPTOR`	Get values from the SQL descriptor area.

Statement	Description
OPEN	Open a prepared cursor.
PREPARE	Compile an SQL source statement into an internal object form.
SET DESCRIPTOR	Set values in the SQL descriptor area.

All statements submitted to dynamic SQL programs must be prepared.

All prepared statements and singleton SELECT statements, where the result set contains only one row, are executed with the EXECUTE statement.

All other SELECT statements and calls to result set procedures are executed using OPEN and FETCH for a cursor declared with the prepared statement.

The declaration of a cursor for a statement, DECLARE CURSOR, must always precede the PREPARE operation for the same statement in an application using dynamic SQL.

SQL Descriptor Area

The SQL descriptor area is used for managing input and output data in dynamically submitted SQL statements containing parameter markers, and for managing result sets (e.g. returned by a SELECT statement.)

An SQL descriptor area is allocated with the ESQL statement ALLOCATE DESCRIPTOR and deallocated with DEALLOCATE DESCRIPTOR. See the *Mimer SQL Reference Manual, Chapter 12, SQL Statements*, for more information.

A program may allocate several separate descriptor areas, identified by different descriptor names. Normally one descriptor is used for input data and one for output data. The describe statement is used to populate an SQL descriptor.

The following statement types can use information from SQL descriptor areas:

- all SELECT statements and calls to result set procedures
- INSERT, DELETE, UPDATE, SET and CALL statements using parameter markers
- ENTER statements.

The following statement types do not use SQL descriptor areas:

- all data definition statements, security control statements, access control statements (except ENTER) and transaction control statements
- INSERT, DELETE, UPDATE and CALL statements using only constant expressions.

In practice, programs using dynamically submitted SQL statements are usually written as though all submitted statements use SQL descriptor areas (since the nature of the submitted statement is not known until run-time).

SQL descriptor areas can be left out of a program only if it is known in advance that they will not be needed (for instance in an application program which will handle only submitted data definition statements).

The Structure of the SQL Descriptor Area

The SQL descriptor area is a storage area holding information about the described statement. It is allocated and maintained with ESQL statements.

It consists of a descriptor header and one or more item descriptor areas. The descriptor header contains two fields, TOP_LEVEL_COUNT and COUNT.

TOP_LEVEL_COUNT is the number of parameters in the descriptor and COUNT is the number of item areas.

The individual fields of the item descriptor area can be accessed with the GET DESCRIPTOR and SET DESCRIPTOR statements. Each descriptor item contains fields for data type, size and scale. The complete list of fields can be seen at *Mimer SQL Reference Manual, Chapter 12, GET DESCRIPTOR*, and *Mimer SQL Reference Manual, Chapter 12, SET DESCRIPTOR*.

Preparing Statements

All statements submitted to dynamic SQL programs must be prepared. The simplest form of the operation uses a PREPARE statement, see the *Mimer SQL Reference Manual, Chapter 12, PREPARE*, for the syntax description. The operation may also be combined with EXECUTE as a simple statement in the shorthand form EXECUTE IMMEDIATE.

The source form of the statement must be contained in a host variable, containing the statement string. (The statement string itself is not preceded by EXEC SQL nor terminated by the language-specific embedded delimiter.)

The prepared form of the statement is named by an SQL-identifier or a host variable, for extended statements, see *Extended Dynamic Cursors* on page *64*.

In the following example the source form of the statement is given as a string constant for illustrative purposes, however, the statement would usually be read from some input source, e.g. the terminal, at run-time:

```
...
EXEC SQL BEGIN DECLARE SECTION;
   string SQL_TXT(255);
   ...
EXEC SQL END DECLARE SECTION;
...

SQL_TXT := "CREATE INDEX pdt_product_search
               ON products(product_search)";
EXEC SQL PREPARE OBJECT FROM :SQL_TXT;
...
```

Extended Dynamic Cursors

A typical cursor is identified by an SQL identifier. An extended cursor makes it possible to represent a dynamic cursor by a host variable or a literal. An extended cursor is allocated by the application with the ALLOCATE CURSOR statement, see the *Mimer SQL Reference Manual, Chapter 12, ALLOCATE CURSOR*, for the syntax description.

When the application is finished with the processing of the SQL statement, the prepared statement may be destroyed by executing the DEALLOCATE PREPARE statement, see the *Mimer SQL Reference Manual, Chapter 12, DEALLOCATE PREPARE*, for the syntax description. DEALLOCATE PREPARE also destroys any extended cursor that was associated with the statement.

Example of how extended cursors are used:

```
...
EXEC SQL BEGIN DECLARE SECTION;
   string SQL_TXT(255);
   string C1(128);
   string STM1(128);
   integer HOSTVAR1;
   string HOSTVAR2(10);
   ...
EXEC SQL END DECLARE SECTION;
...

SQL_TXT := "SELECT col1, col2
              FROM tab1";
STM1 := "STMT_1";
EXEC SQL PREPARE :STM1 FROM :SQL_TXT;

C1 := "CUR_1";
EXEC SQL ALLOCATE :C1 CURSOR FOR :STM1;
...

EXEC SQL ALLOCATE DESCRIPTOR 'RESDESC' WITH MAX 50;
EXEC SQL DESCRIBE OUTPUT :STM1 USING SQL DESCRIPTOR 'RESDESC';
...
EXEC SQL OPEN :C1;
EXEC SQL WHENEVER NOT FOUND GOTO done;

loop
   EXEC SQL FETCH :C1
               INTO SQL DESCRIPTOR 'RESDESC';
   EXEC SQL GET DESCRIPTOR 'RESDESC' VALUE 1 :HOSTVAR1 = DATA;
   EXEC SQL GET DESCRIPTOR 'RESDESC' VALUE 2 :HOSTVAR2 = DATA;
   ...
   display HOSTVAR1, HOSTVAR2, ...;
end loop;

done:
EXEC SQL CLOSE :C1;
EXEC SQL DEALLOCATE DESCRIPTOR 'RESDESC';
EXEC SQL DEALLOCATE PREPARE :STM1;
...
```

Describing Prepared Statements

Statements returning a result set and statements containing parameter markers can be described to obtain information about the number and data types of the parameters.

There are two forms of `DESCRIBE`:

- `DESCRIBE OUTPUT` for result set values
- `DESCRIBE INPUT` for input and output parameters.

Both forms of `DESCRIBE` use the object (prepared) form of the statement as an argument. The same statement may be described in both senses if necessary.

For example:

```
EXEC SQL BEGIN DECLARE SECTION;
   string SQLA1(128);
   integer MAXOCC;
   string SOURCE(255);
EXEC SQL END DECLARE SECTION;
...

MAXOCC := 15;
SQLA1 := "SQL_AREA_1";
EXEC SQL ALLOCATE DESCRIPTOR :SQLA1 WITH MAX 20;
EXEC SQL ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX :MAXOCC;
...

EXEC SQL PREPARE 'OBJECT' FROM :SOURCE;
EXEC SQL DESCRIBE OUTPUT 'OBJECT' USING SQL DESCRIPTOR :SQLA1;
EXEC SQL DESCRIBE INPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA2';
...
```

`DESCRIBE` places information about the prepared statement in the SQL descriptor areas. See *SQL Descriptor Area* on page *63* for a description of the SQL descriptor area.

The contents of the SQL descriptor area is read with the `GET DESCRIPTOR` statement and updated with the `SET DESCRIPTOR` statement.

Describing Output Variables

The items in the result set for a statement are described with the `DESCRIBE OUTPUT` statement. The keyword `OUTPUT` may be omitted.

The `DESCRIBE OUTPUT` statement shows:

- whether the statement returns a result set or not. This is indicated by the value of the `COUNT` field of the SQL descriptor area which is set to zero for statements that do not return a result set. Statements that return a result set are calls to result set procedures, see *Result Set Procedures* on page *264*, and select-expressions (refer to the *Mimer SQL Reference Manual, Chapter 12, SELECT*).
- dynamic SQL programs must test for this after each `DESCRIBE` operation because the treatment of statements that return result sets differs from the treatment of those that do not, see *Handling Prepared Statements* on page *67*. If the statement returns a result set, the `DESCRIBE` statement will place information about the items in the result set in the fields of the descriptor area.
- whether the current descriptor area allocation is sufficient or not. Insufficient area is indicated by the `SQLSTATE` variable set to a warning state and a value of `COUNT` (required number of items) greater than that specified in the `WITH MAX …` clause of the `ALLOCATE DESCRIPTOR` statement, or greater than 100 if no `WITH MAX …` clause was specified. If the area is insufficient, no items are described.

Describing Input Variables

The `DESCRIBE INPUT` statement is used to describe parameter markers.

The value of the `COUNT` field of the SQL descriptor area indicates the number of parameter markers in the statement (a value of zero indicates no input parameters). A value greater than that specified in `WITH MAX …` indicates that the allocated SQL descriptor area is too small and the describe operation will not be performed. This situation is handled as described above for `DESCRIBE OUTPUT`.

Note: If the prepared statement is a call to a stored procedure that uses parameter markers, these will be described by the DESCRIBE INPUT statement. This is regardless of how the formal parameter is specified in the procedure definition. Whether the parameter is IN, INOUT or OUT can be seen from the PARAMETER_MODE field in the descriptor area.

Handling Prepared Statements

After PREPARE and DESCRIBE, the way in which submitted statements are handled differs according to whether the statement is executable or whether it returns a result set.

- **Executable statements** are executed using the EXECUTE statement, with the object (prepared) form of the submitted statement as the argument.
- **Result set statements**, a cursor is used for these statements, associated with the object form of the prepared statement and are executed with OPEN and FETCH.

Executable Statements

Executable statements are identified by a value of zero in the COUNT field of the SQL descriptor area after a DESCRIBE OUTPUT statement. If the statement does not contain any parameter markers, it may be executed directly.

If, on the other hand, the statement contains parameter markers, the statement must be executed with an SQL descriptor area for input and output values.

Note: All parameter markers used in a call statement are described with the DESCRIBE INPUT statement, regardless of the mode of the formal parameter.

Parameter markers must be used for all INOUT or OUT parameters when a call statement is prepared dynamically.

The descriptor areas referenced in the EXECUTE statement may be replaced by explicit lists of host variables, provided that the number and data types of the user variables in the source statement are known when the program is written (so that variables can be declared and the appropriate variable list written into the EXECUTE statement).

This facility is of limited use, since the occasions when the user constructs freely chosen SQL statements with a predetermined number of user variables are rare.

EXECUTE IMMEDIATE

The shorthand form EXECUTE IMMEDIATE combines the functions of PREPARE and EXECUTE. This form may only be used for executable statements known to have no parameter markers and is therefore of value only in contexts where the user is restricted to this type of statement. (Data definition and security control statements fall into this category, since user variables are not permitted in the syntax of these statements. EXECUTE IMMEDIATE can therefore be useful for application programs designed specifically to handle database definition statements).

Example

```
sprintf(ddlstr, "drop ident %s cascade", str);
exec sql EXECUTE IMMEDIATE :ddlstr;
```

Result Set Statements

Statements returning a result set are identified by a non-zero value in the COUNT field of the SQL descriptor area after DESCRIBE OUTPUT.

Dynamically submitted SELECT statements and calls to result set procedures are handled through cursors. Cursors are declared or allocated for the object (prepared) form of submitted result set returning statements.

Note: A DECLARE CURSOR statement must precede the PREPARE statement in the program code. If ALLOCATE CURSOR is used instead of DECLARE CURSOR, the statement must have been prepared before the cursor can be allocated. The SQL statement must also be prepared before the cursor is opened.

If the source form of the result set returning statement contains parameter markers, these must be described before the cursor is opened and the OPEN statement must reference the relevant descriptor area. In the rare case where the number and data type of the user variables are known when the program is first written, the OPEN statement may reference an explicit variable list instead of a descriptor area.

The descriptor area used for the submitted result set returning statement is referenced when data is retrieved with the FETCH statement.

Example

```
...
EXEC SQL ALLOCATE DESCRIPTOR 'SQLA1' WITH MAX 30;
EXEC SQL ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX 30;
...

EXEC SQL PREPARE 'OBJECT' FROM :SOURCE;
...

EXEC SQL DESCRIBE OUTPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA1';
EXEC SQL GET DESCRIPTOR 'SQLA1' :NO_OUT = COUNT;
if NO_OUT = 0 then
   RESULT_SET := FALSE;
else
   EXEC SQL ALLOCATE 'C1' CURSOR FOR 'OBJECT';
   RESULT_SET := TRUE;
end if;
...

EXEC SQL DESCRIBE INPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA2';
EXEC SQL GET DESCRIPTOR 'SQLA2' :NO_IN = COUNT;
...

if RESULT_SET then
   EXEC SQL OPEN 'C1' USING SQL DESCRIPTOR 'SQLA2';
   ...
   EXEC SQL FETCH 'C1' INTO SQL DESCRIPTOR 'SQLA1';
   ...
else
   ...
end if;
```

Example Framework for Dynamic SQL Programs

This section gives a general framework (in pseudo code) for dynamic SQL programs designed to handle any valid SQL statement as input. The framework is largely a synthesis of the example fragments given earlier in this chapter.

The framework is written as a single sequential module to emphasize the order of operations.

Host variable declarations are omitted. Handling of values returned by FETCH is also omitted.

Example Framework

```
-- Allocate two SQL descriptor areas
EXEC SQL ALLOCATE DESCRIPTOR 'SQLA1' WITH MAX 50;
EXEC SQL ALLOCATE DESCRIPTOR 'SQLA2' WITH MAX 50;

-- read statement from terminal
read INPUT into SOURCE;

-- prepare statement
EXEC SQL PREPARE 'OBJECT' FROM :SOURCE;

-- describe statement and set type/parameter usage flags
EXEC SQL DESCRIBE OUTPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA1';
EXEC SQL GET DESCRIPTOR 'SQLA1' :NO_OUT = COUNT;
if NO_OUT = 0 then
   RESULT_SET := FALSE;
else
   -- allocate cursor for result set
   EXEC SQL ALLOCATE 'C1' CURSOR FOR 'OBJECT';
   RESULT_SET:= TRUE;
end if;

EXEC SQL DESCRIBE INPUT 'OBJECT' USING SQL DESCRIPTOR 'SQLA2';
EXEC SQL GET DESCRIPTOR 'SQLA2' :NO_IN = COUNT;
-- execute statement or open cursor and fetch after assigning
-- values to input variables
if RESULT_SET then
   EXEC SQL OPEN 'C1' USING SQL DESCRIPTOR 'SQLA2';
   loop
      EXEC SQL FETCH 'C1' INTO SQL DESCRIPTOR 'SQLA1';
      exit when NO_MORE_REQUIRED or SQLSTATE = "02000";
      ... -- process results of FETCH
   end loop;

   EXEC SQL CLOSE 'C1';
else
   EXEC SQL EXECUTE 'OBJECT' USING SQL DESCRIPTOR 'SQLA2';
end if;

EXEC SQL DEALLOCATE PREPARE 'OBJECT';
```

Note: Features that are specific to real host languages are described in *Host Language Dependent Aspects* on page *307*.

Handling Errors and Exceptions

Errors may arise at three general levels in an embedded SQL (ESQL) program (not counting errors in the SQL-independent host language code). These are syntax, semantic and run-time errors.

See *Managing Exception Conditions* on page *267* for information about managing exception conditions in routines and triggers.

Syntax Errors

Syntax errors are constructions that break the rules for formulating SQL statements. For example:

- **Spelling errors in keywords:**

  `SLEECT` instead of `SELECT`

- **Incorrect or missing delimiters:**

  `DELETEFROM` instead of `DELETE FROM`

  `SELECT column1;column2` instead of `SELECT column1,column2`

- **Incorrect clause ordering**

  `UPDATE … WHERE … SET` instead of `UPDATE … SET … WHERE`

The preprocessor does not accept syntactically incorrect statements. The error must be corrected before the program can be successfully preprocessed.

Semantic Errors

Semantic errors arise when SQL statements are formulated in full accordance with the syntax rules, but do not reflect the programmer's intentions correctly.

Some semantic errors, e.g. incorrect references to database objects, are detected and reported by the ESQL preprocessor but other semantic errors will not become apparent until run-time.

Run-time Errors

Run-time errors and exception conditions (for example warnings) arising during execution of ESQL statements are signaled by the contents of the `SQLSTATE` status variable described in *The SQLSTATE Variable* on page *49*. A list of possible `SQLSTATE` values is provided in *SQLSTATE Return Codes* on page *323*.

The `GET DIAGNOSTICS` statement can be used to retrieve detailed information about an exception, see the *Mimer SQL Reference Manual, Chapter 12, GET DIAGNOSTICS*, for the syntax description.

The `NATIVE_ERROR` and `MESSAGE_TEXT` fields of the diagnostics area retrieved by using `GET DIAGNOSTICS` are used to get the internal Mimer SQL return code (aka `SQLCODE`) and the descriptive text, respectively, relating to the exception, these are listed in *Native Mimer SQL Return Codes* on page *329*.

Testing for Run-time Errors and Exception Conditions

The application program may test the outcome of a statement in one of two ways:

- by explicitly testing the contents of the `SQLSTATE` variable
- by using the SQL statement `WHENEVER`, see the *Mimer SQL Reference Manual, Chapter 12, WHENEVER* for the syntax description, which tests the class of the SQLSTATE variable.

An application program may contain any number of `WHENEVER` statements, and the statements may be placed anywhere in the program. A separate `WHENEVER` statement must be issued for each situation (`NOT FOUND`, `SQLEXCEPTION` or `SQLWARNING`) which is to be tested.

When an exception condition arises, action will be taken as specified in the `WHENEVER` statement most recently encountered in the code, for the respective condition.

`WHENEVER` statements are expanded by the preprocessor into explicit tests. These tests are placed after every subsequent SQL statement in that program until a new `WHENEVER` statement is issued for the same condition.

Two important consequences follow:

- `WHENEVER` statements are preprocessed strictly in the order in which they appear in the source code, regardless of execution order or conditional execution that the source code might imply.

  For instance, the `WHENEVER` statement in the following Fortran construction is expanded by the preprocessor, even though its execution is never actually requested:

```
      ...
       GOTO 1025
       EXEC SQL WHENEVER SQLEXCEPTION GOTO 1600
1025   CONTINUE
       EXEC SQL DELETE FROM MYTABLE
      ...
```

- Mixing explicit tests and `WHENEVER` statements requires care. As a general rule, it is advisable to use either hand-written tests or `WHENEVER` statements in a program module, and to avoid mixing them.

  The condition handling defined by a `WHENEVER` statement applies to the SQL statements that follow it in the source code. If a `GOTO` action is defined, the pre-processor inserts an exception test and action directly after each SQL statement affected by it and thus before any hand-written tests in the source code. The hand-written test in this situation would never be executed.

  If `CONTINUE` is specified in a `WHENEVER` statement, the pre-processor does not insert an exception test and action, thus no exception handling is defined by the `WHENEVER` statement. Any hand-written tests present in the source code will then take effect.

  The interchange between hand-written exception handling and the implicit exception handling inserted by the pre-processor (or not) can be confusing. It is therefore advisable to make a clear coding decision to use one method or the other.

Example using WHENEVER SQLEXCEPTION

```
#include <stdlib.h>
#include <wchar.h>

int main()
{
  exec sql BEGIN DECLARE SECTION;
  char sqlstate[6];
  nchar varying dbname[129];
  nchar varying dbtype[129];
  exec sql END DECLARE SECTION;

  exec sql WHENEVER SQLEXCEPTION GOTO get_diagn;

  exec sql CONNECT TO 'db' USER 'username' USING 'password';

  exec sql DECLARE c CURSOR FOR
           select databank_name, databank_type
           from information_schema.ext_databanks;

  exec sql WHENEVER NOT FOUND GOTO end_of_table;

  exec sql OPEN c;

  while (1)
  {
      exec sql FETCH c INTO :dbname, :dbtype;
      wprintf(L"Databank: %ls\nType:     %ls\n\n", dbname, dbtype);
  }

end_of_table:
  exec sql CLOSE c;

  exec sql COMMIT;
  exec sql DISCONNECT ALL;
  exit(0); /* Exit with success */


get_diagn:
/* print diagnostics message(s) for the most recent statement */
  {
    exec sql BEGIN DECLARE SECTION;
    int i;
    int exceptions;
    int errcode;
    nchar varying message[255];
    exec sql END DECLARE SECTION;

    exec sql WHENEVER SQLEXCEPTION CONTINUE;

    exec sql GET DIAGNOSTICS :exceptions = NUMBER; /* How many exceptions? */
    for (i=1; i<=exceptions; i++) {
      exec sql GET DIAGNOSTICS EXCEPTION :i
        :message = MESSAGE_TEXT, :errcode = NATIVE_ERROR;
      wprintf(L"(%d)  %ls\n", errcode, message);
    }
    exec sql ROLLBACK;
    exec sql DISCONNECT;
    exit(-1); /* Error exit */
  }
}
```

Example using explicit return code checking

```
#include <stdlib.h>
#include <wchar.h>
void get_diagn();

int main()
{
  exec sql BEGIN DECLARE SECTION;
  int sqlcode;
  nchar varying dbname[129];
  nchar varying dbtype[129];
  exec sql END DECLARE SECTION;

  exec sql CONNECT TO 'db' USER 'username' USING 'password';
  if (sqlcode != 0) get_diagn();

  exec sql DECLARE c CURSOR FOR
           select databank_name, databank_type
           from information_schema.ext_databanks;

  exec sql OPEN c;
  if (sqlcode != 0) get_diagn();

  while (sqlcode == 0)
  {
      exec sql FETCH c INTO :dbname, :dbtype;
      if (sqlcode < 0) get_diagn();
      if (sqlcode != 100)
        wprintf(L"Databank: %ls\nType:     %ls\n\n", dbname, dbtype);
  }

  exec sql CLOSE c;
  if (sqlcode != 0) get_diagn();

  exec sql COMMIT;
  if (sqlcode != 0) get_diagn();

  exec sql DISCONNECT ALL;
  if (sqlcode != 0) get_diagn();

  exit(0); /* Exit with success */
}

void get_diagn()
/* print diagnostics message(s) for the most recent statement */
{
  exec sql BEGIN DECLARE SECTION;
  int i;
  int exceptions;
  int errcode;
  nchar varying message[255];
  int sqlcode;
  exec sql END DECLARE SECTION;


  exec sql GET DIAGNOSTICS :exceptions = NUMBER; /* How many exceptions? */
  for (i=1; i<=exceptions; i++) {
    exec sql GET DIAGNOSTICS EXCEPTION :i
        :message = MESSAGE_TEXT, :errcode = NATIVE_ERROR;
    wprintf(L"(%d)  %ls\n", errcode, message);
  }
  exec sql ROLLBACK;
  exec sql DISCONNECT;
  exit(-1); /* Error exit */
}
```

Chapter 5

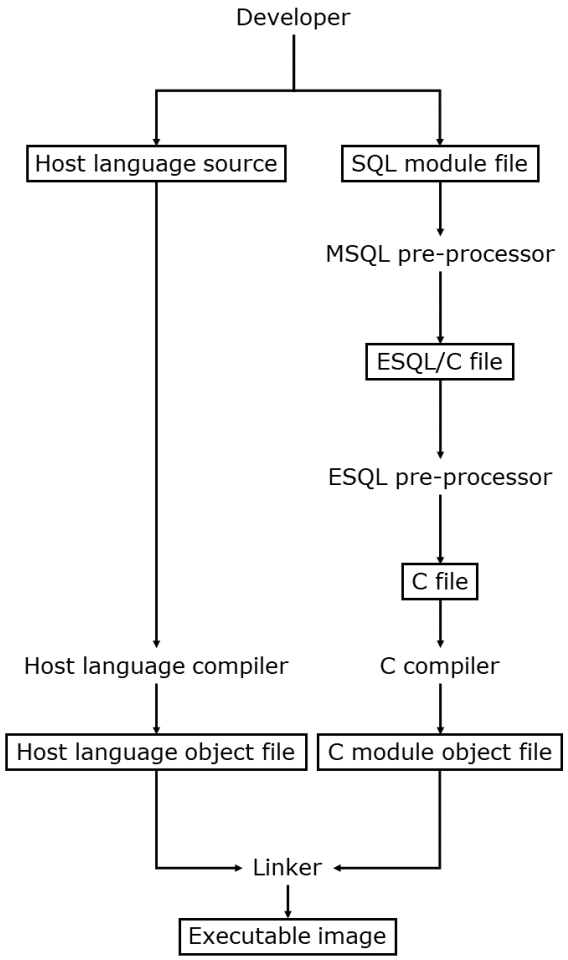
Module SQL

This chapter discusses the scope, principles, processing and structure of Module SQL (MSQL).

MSQL enables you to call SQL statements in a host program written in C/C++, COBOL, Fortran or Pascal, without embedding the actual SQL statements in the host program. The SQL statements are explicitly put into a separate SQL module, that is written in the Module language and maintained separately from the host program.

The Scope of Mimer Module SQL

The Mimer MSQL files are pre-processed through the Mimer Module SQL pre-processor into Embedded C code. The resulting Embedded C code can then be processed through the Mimer Embedded SQL pre-processor into C code which can be compiled, and then linked together with the host application.



The SQL statements supported are therefore limited to the SQL statements supported by Embedded SQL, which are described in *Chapter 4, Embedded SQL*.

General Principles for SQL Modules

The following sections discuss host languages, pre-processors, identifying SQL statements, code, comments and recommendations.

Host languages

You can call the C code that comes out of the MSQL pre-processor from any host language that supports the used data types. Mimer MSQL supports C/C++, COBOL, Fortran and Pascal.

Information given in this manual applies to all languages unless otherwise explicitly stated. Language-specific information is detailed in *Host Language Dependent Aspects* on page *91*.

Writing an SQL module

An SQL module is a list of SQL statements, where each SQL statement is included in a procedure. Each procedure contains only one SQL statement. The MSQL syntax is case insensitive. Routine, cursor and parameter names (except SQLCODE and SQLSTATE) are case sensitive, as they will follow to the generated C code which is case sensitive.

Module declaration

The syntax for declaring an SQL module is as follows:

```
MODULE module-name
LANGUAGE (C|COBOL|FORTRAN|PASCAL)
```

The `LANGUAGE` clause tells the module which host language it will be called from and adapts some data types to the given host language. For a full list of how a host application should handle all Mimer SQL data types, see *Host Language Dependent Aspects* on page *91*.

LANGAUGE C

- The `CHARACTER`, `VARCHAR`, `BOOLEAN`, `DATETIME` and `INTERVAL` SQL data types are handled like a null-terminated C char array in the generated MSQL C code. When calling module procedures, all host languages need to make room for and apply null-termination. The Fortran `CHARACTER` data type cannot be used. This includes SQLSTATE.
- The `CLOB` SQL data type is handled like a C struct specified according to the SQL standard in the generated MSQL C code. When calling module procedures, all host languages need to use corresponding data structures.
- The `DECIMAL` SQL data type is handled like a C `double` in the generated MSQL C code. When calling module procedures, all host languages need to use corresponding data types. The COBOL `PACKED DECIMAL (COMP-3)` data type cannot be used.

LANGUAGE FORTRAN

- The `CHARACTER`, `VARCHAR`, `BOOLEAN`, `DATETIME` and `INTERVAL` SQL data types are handled like a fixed length Fortran `CHARACTER` array in the generated MSQL C code, without null termination. This includes SQLSTATE.
- The `CLOB` SQL data type is handled like a fixed length Fortran `CHARACTER` array in the generated MSQL C code, without null termination.
- The `DECIMAL` SQL data type is handled like a C `double` in the generated MSQL C code. When calling module procedures, the Fortran application needs to use the corresponding data type.

LANGUAGE COBOL

- The `CHARACTER`, `VARCHAR`, `BOOLEAN`, `DATETIME` and `INTERVAL` SQL data types are handled like a fixed length COBOL `PICTURE X` array in the generated MSQL C code, without null termination. This includes SQLSTATE.

- The CLOB SQL data type is handled like a fixed length COBOL PICTURE X array in the generated MSQL C code, without null termination.
- The DECIMAL SQL data type is handled like a COBOL PACKED DECIMAL (COMP-3) data type of the expected precision and scale in the generated MSQL C code.

LANGUAGE PASCAL

- The CHARACTER, VARCHAR, BOOLEAN, DATETIME and INTERVAL SQL data types are handled like a fixed length Pascal PACKED ARRAY OF CHAR in the generated MSQL C code, without null termination. This includes SQLSTATE.
- The CLOB SQL data type is handled like a fixed length COBOL Pascal PACKED ARRAY OF CHAR in the generated MSQL C code, without null termination.
- The DECIMAL SQL data type is handled like a C double in the generated MSQL C code. When calling module procedures, the Pascal application needs to use the corresponding data type.

The SQLCODE parameter is always translated to a pointer to an integer, no matter the chosen module language.

Cursor declarations

All cursors that are to be used in the module's procedures need to be declared before any procedure is declared. A cursor is declared using the corresponding SQL syntax as follows:

```
DECLARE cursor-name CURSOR FOR cursor-statement
```

A module can contain multiple cursor declarations. They all need to be declared before the module procedures, and they are delimited by new rows. The cursor declaration can be split on more than one row wherever a white-space is permitted.

```
DECLARE cursor-name1 CURSOR FOR
cursor-statement1
DECLARE cursor-name2
CURSOR FOR
cursor-statement2
```

Module procedures

Following the module and cursor declarations, the procedures that execute SQL statements are declared. An SQL module language procedure has a name, parameter declarations, and an executable SQL statement.

The syntax for declaring an SQL module procedure is as follows:

```
PROCEDURE procedure-name
    [parameter-declaration
    [, parameter-declaration];
SQL statement;
```

The syntax for the parameter declaration is as follows:

```
:parameter-name datatype
```

or

```
SQLSTATE
```

or

```
SQLCODE
```

Example:

```
PROCEDURE fetch_data
    :param_1 INTEGER
    :param_2 SMALLINT
    SQLSTATE;
FETCH data_cursor INTO
    :param_1,
    :param_2;
```

The parameters may be input parameters, output parameters or both.

Parameters are separated by commas or whites paces and ended with semicolon.

SQLSTATE and SQLCODE parameters are status parameters through which errors are passed. Use either, not both, in the same SQL module. Using both in the same SQL module causes undefined behavior.

Comments

You can comment your SQL module file with double dashes, as follows:

```
-- A very useful cursor.
DECLARE cursor-name CURSOR FOR cursor-statement
```

Comments are not forwarded to the embedded C and C code.

Recommendations

We recommend the following when writing MSQL:

- Avoid variable names beginning with the letters `SQL` (except for SQLSTATE and SQLCODE, which should be used when appropriate).
- Avoid procedure names ending with a number.
- Avoid separating identifiers (e.g. procedure or parameter names) with casing only. Even though the generated C code might be valid, the calling host language might be case insensitive.

Calling an SQL module

In the host program, you call an SQL procedure at whatever point in the host program you want to execute the SQL statement in that procedure. You call the SQL procedure as if it was a subprogram in C.

The following SQL module procedure:

```
PROCEDURE fetch_data
    :param_1 INTEGER
    :param_2 SMALLINT
    SQLSTATE;
FETCH data_cursor INTO
    :param_1,
    :param_2;
```

will be processed into the following C function:

```
void fetch_data(int* param_1, short* param_2, char sqlstate[6])
```

When calling this function from your host program, the parameters you send need to be compatible with the C data types in the C function. Input char arrays might need to be null-terminated, as per C standard, see Writing an SQL module - Module declaration and *Host Language Dependent Aspects* on page *91* for more details.

Processing MSQL

The following sections discuss pre-processing and processing MSQL.

Pre-processing - the MSQL command

An SQL module file must first be pre-processed using the MSQL command, then pre-processed using the ESQL command (see *Processing ESQL* on page *35*), before it is compiled with the C compiler.

The input to the pre-processor is thus an SQL module written in Module language.

The output from the preprocessor is always an Embedded C source code file, containing both C and ESQL statements. If the host language is C, an Embedded C header file is also generated.

The default file extensions for preprocessor input and output files are shown in the table below:

Input file extension	Output file extension	Header file extension
`.msq`	`.ec`	`.eh`

Invoking the MSQL Preprocessor

The MSQL preprocessor has the following command line syntax:

```
msql [-n] [-i enc] [-o enc] [-b yes|no] [-u yes|no] infile
    [outfile [headerfile]]

msql [--nologo] [--inencoding=enc] [--outencoding=enc] [--inbom=yes|no]
    [--outbom=yes|no] infile [outfile [headerfile]]

msql [-v|--version] | [-?|--help]
```

Options

Unix-style	VMS-style	Function
`-n` `--nologo`	`/NOLOGO`	Suppresses the display of the copyright message on the screen (warnings and errors are always displayed on the screen.)

Unix-style	VMS-style	Function

`-i` *encoding*
`--inencoding=`*encoding*
	`/INENCODING=`*encoding*	The encoding to read the input file in.
		Valid options are:
		`default`
		`latin1`
		`UTF8`
		`UTF16`
		`UTF16BE`
		`UTF16LE`
		`UTF32`
		`UTF32BE`
		`UTF32LE`
		The `default` encoding is platform and locale dependent.
		Sets to `default` if omitted.

`-o` *encoding*
`--outencoding=`*encoding*
	`/OUTENCODING=`*encoding*	The encoding to write the output file in.
		Valid options are:
		`default`
		`latin1`
		`UTF8`
		`UTF16`
		`UTF16BE`
		`UTF16LE`
		`UTF32`
		`UTF32BE`
		`UTF32LE`
		The `default` encoding is platform and locale dependent.
		Equals the input file encoding if omitted.

`-b yes/no`
`--inbom=yes/no`
	`/INBOM=yes/no`	If BOM should explicitly be used/not used when reading from the input file.
		Valid options are:
		`yes`
		`no`
		Uses the `inencoding`'s default if omitted.

Unix-style	VMS-style	Function

`-u yes/no`
`--outbom=yes/no`
	`/OUTBOM=yes/no`	If BOM should explicitly be used/not used when writing to the output file.
		Valid options are:
		`yes`
		`no`
		Uses the `outencoding`'s default if omitted.

`infile`	`infile`	The input-file containing the SQL module source code to be pre-processed. If no file extension is specified, the .msq file extension is assumed (previously described in this section.)
`[outfile]`	`[outfile]`	The output-file which will contain the Embedded C code generated by the preprocessor. If not specified, the output file will have the same name as the input file, but with the .ec file extension (previously described in this section.)
`[headerfile]`	`[headerfile]`	The header-file which will contain the Embedded C header generated by the preprocessor. If not specified, the header file will have the same name as the output file, but with the .eh file extension (previously described in this section.) If the input file has a target host language other than C, this argument will not be used.

Note: As an application programmer, you should never attempt to directly modify the output from the preprocessor.

Any changes that may be required in an SQL module should be introduced into the original SQL module code. Mimer Information Technology AB cannot accept any responsibility for the consequences of modifications to the pre-processed code.

What Does the Preprocessor Do?

The preprocessor checks the syntax (See *Handling errors and exceptions on page 88* for a more detailed discussion of how errors are handled.) Syntactically invalid statements cannot be pre-processed and the source code must be corrected.

Processing MSQL

Compiling

The output from the MSQL and ESQL preprocessors is compiled in the usual way using the appropriate C compiler and linked with the appropriate routine libraries.

Linux: On Linux platforms, the gcc compiler is supported.

VMS: The VSI C compiler and the HP C compiler are supported on the OpenVMS platform.

Win: On Windows platforms, the C compiler identified by the `cc` symbol in the file `.\dev\samples\makefile_msql.mak` below the installation directory is supported.

Note: Other compilers, from other software distributors, may or may not be able to compile the MSQL and ESQL preprocessor output. Mimer Information Technology cannot guarantee the result of using a compiler that is not supported.

Linking

Linking the processed SQL module with a host application is done in the same way as user-written embedded C code, see *Linking Applications* on page *41*.

The SQL Compiler

At run-time, database management requests are passed to the SQL compiler responsible for implementing the SQL functions in the application program.

The SQL compiler performs two functions:

- It checks SQL statements semantically against the data dictionary.
- It optimizes operations performed against the database (i.e. internal routines determine the most efficient way to execute the SQL request, with regard to the existence of secondary indexes and the number of rows in the tables addressed by the statement). You, as a programmer, do not need to worry, for instance, about the order in which tables are addressed in a complex selection condition. This optimization process is completely transparent.

Note: Since all SQL statements are compiled at run-time, there can be no conflict between the state of the database at the times of compilation and execution. Moreover, the execution of SQL statements is always optimized with reference to the current state of the database.

Connecting to a Database

Connecting to a database is done much in the same way as in embedded programs, see *Connecting to a Database* on page *42*. Every connect attempt is a separate SQL statement that needs its own module procedure.

Examples

Example SQL module:

```
PROCEDURE connect_ident
    :ident VARCHAR(100)
    :pswd VARCHAR(100)
    SQLSTATE;
CONNECT TO 'the_database' USER :ident USING :pswd;
```

Example host application in C:

```
#include "module_name.h"

int main()
{
    char sqlstate[6];
    char ident[101] = "SYSADM";
    char pswd[101] = "SYSADM";

    connect_ident(ident, pswd, &sqlcode);
```

Example host application in Fortran:

```
      CHARACTER*5 SQLSTATE
      CHARACTER*100 IDENT
      CHARACTER*100 PSWD

      IDENT = 'SYSADM'
      PSWD = 'SYSADM'
      CALL CONNECT_IDENT(IDENT, PSWD, SQLSTATE)
```

Example host application in Cobol:

```
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 SQLSTATE PIC X(5).
       01 IDENT PIC X(100).
       01 PSWD PIC X(100).

       PROCEDURE DIVISION.
       HEAD SECTION.

       MAIN.

           MOVE "SYSADM" TO IDENT.
           MOVE "SYSADM" TO PSWD.

           CALL "CONNECT_IDENT" USING IDENT, PSWD, SQLSTATE.
```

Example host application in Pascal:

```
type chararray = packed array [1..100] of char;
type sqlstatearray = packed array [1..5] of char;

var
sqlstate : sqlstatearray;
ident : chararray;
pswd : chararray;

procedure connect_ident(var ident : chararray, var pswd : chararray,
   var sqlstate: sqlstatearray); external;

begin
   ident := 'SYSADM';
   pswd := 'SYSADM';
   connect_ident(ident, pswd, sqlstate);
```

Communicating with the Application Program

Information is transferred between the host application program and the Mimer SQL database manager through an SQL module much in the same way as in embedded SQL programs, see *Communicating with the Application Program* on page *46*.

Indicator variables

In MSQL, as in ESQL, indicator variables associated with main variables are used to handle null values in database tables. Indicator variables are declared in the parameter list to a module procedure like the main parameters, with the SQL data type SMALLINT. It is translated to the C type short. The host language then provides the variable in the parameter list in the same way as the main variables.

```
PROCEDURE fetch_name
    :name VARCHAR(20)
    :name_indicator SMALLINT
    SQLSTATE;
FETCH name_cursor INTO :name:name_indicator;
```

Accessing data

This section explains how SQL modules retrieve data.

Retrieving data using cursors

Data is retrieved much in the same way as in ESQL, see *Accessing Data* on page *50*. The host application will have to check the value of SQLCODE or SQLSTATE in order to find out when the cursor has reached the end of the result set.

Examples

Example SQL module:

```
DECLARE currencies_cursor CURSOR FOR
SELECT code FROM mimer_store.currencies

PROCEDURE open_currencies_cursor
    SQLSTATE;
OPEN currencies_cursor;

PROCEDURE fetch_currency_code
    :code CHARACTER(3)
    SQLSTATE;
FETCH currencies_cursor INTO :code;

PROCEDURE close_currencies_cursor
    SQLSTATE;
CLOSE currencies_cursor;
```

Example host application in C:

```
#include "module_name.h"

int main()
{
    int sqlcode;
    char code[4];
    open_currencies_cursor(&sqlcode);

    while (sqlcode == 0)
    {
        fetch_currency_code(code, &sqlcode);
    }

    close_currencies_cursor(&sqlcode);
```

Example host application in Fortran:

```
      INTEGER*4 SQLCODE
      CHARACTER*3 CODE

      CALL OPEN_CURRENCIES_CURSOR(SQLCODE)

      DO WHILE (SQLCODE .EQ. 0) THEN
          CALL FETCH_CURRENCY_CODE(CODE, SQLCODE)
      END DO

      CALL CLOSE_CURRENCIES_CURSOR(SQLCODE)
```

Example host application in Cobol:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SQLCODE PIC S9(9) USAGE IS BINARY.
01 CODE PIC X(3).

PROCEDURE DIVISION.
HEAD SECTION.

MAIN.

    CALL "OPEN_CURRENCIES_CURSOR" USING SQLCODE.

    PERFORM UNTIL SQLCODE IS NOT ZERO
        CALL "FETCH_CURRENCY_CODE" USING NAME, SQLCODE.
    END-PERFORM.

    CALL "CLOSE_CURRENCIES_CURSOR" USING SQLCODE.
```

Example host application in Pascal:

```
type currency_code_type = packed array [1..3] of char;

var
sqlcode : integer;
code : currency_code_type;

procedure open_currencies_cursor(var sqlcode : integer); external;
procedure fetch_currency_code(var code : currency_code_type,
   var sqlcode : integer); external;
procedure close_currencies_cursor(var sqlcode : integer); external;

begin
   open_currencies_cursor(sqlcode);

   while sqlcode = 0 do
   begin
      fetch_currency_code(code, sqlcode);
   end

   close_currencies_cursor(sqlcode);
```

Retrieving single rows

See *Accessing Data* on page *50* for detailed information on how retrieving single rows work. It is done in an SQL module by declaring a procedure for the specific statement.

Dynamic SQL

Dynamic SQL can be used in the same way as in ESQL, see *Dynamic SQL* on page *60*. The `PREPARE` and `EXECUTE` operations are performed through module procedures.

```
PROCEDURE prepare_stmnt
    :stmnt VARCHAR(300)
    SQLCODE;
PREPARE statement1 FROM :stmnt;

PROCEDURE execute_prepared
    SQLCODE;
EXECUTE statement1;
```

Handling errors and exceptions

Errors may arise at three general levels in an SQL module. These are syntax, semantic and run-time errors.

Syntax Errors

Syntax errors are constructions that break the rules for formulating SQL statements. For example:

- **Spelling errors in keywords:**

  `SLEECT` instead of `SELECT`

- **Incorrect or missing delimiters:**

  `DELETEFROM` instead of `DELETE FROM`

  `SELECT column1;column2` instead of `SELECT column1,column2`

- **Incorrect clause ordering**

  `UPDATE … WHERE … SET` instead of `UPDATE … SET … WHERE`

The preprocessor does not accept syntactically incorrect statements. The error must be corrected before the program can be successfully preprocessed.

Semantic Errors

Semantic errors arise when SQL statements are formulated in full accordance with the syntax rules, but do not reflect the programmer's intentions correctly. Semantic errors are not detected by the MSQL preprocessor.

Run-time Errors

Run-time errors and exception conditions (for example warnings) arising during execution of SQL module procedures are signaled in the same was as in ESQL, see *Handling Errors and Exceptions* on page *69*.

The errors cannot be caught with `WHENEVER` statements that are used in ESQL, the host application has to rely on the values of SQLCODE and SQLSTATE.

Examples

Example SQL module:

```
PROCEDURE connect_sysadm
    SQLCODE;
CONNECT TO '' USER 'SYSADM' USING 'SYSADM';

PROCEDURE get_diagn
    :errcode INT
    :errmsg  VARCHAR(300)
    SQLCODE;
GET DIAGNOSTICS EXCEPTION 1
    :errcode = native_error,
    :errmsg  = message_text;
```

Example host application in C:

```
void connect_sysadm(int* sqlcode);
void get_diagn(int* errcode, char errmsg[301], int* sqlcode);

int main()
{
    int sqlcode;
    int errcode;
    char errmsg[301];

    connect_sysadm(&sqlcode);
    if (sqlcode != 0)
    {
        printf("Failed to connect SYSADM.\n");

        get_diagn(&errcode, errmsg, &sqlcode);
        if (sqlcode != 0)
        {
            printf("Failed to get diagnostics message.\n");
        }
        else
        {
            printf("errcode: %d\n", errcode);
            printf("errmsg: %s\n", errmsg);
        }
    }
```

Example host application in Fortran:

```
      INTEGER*4 SQLCODE
      INTEGER*4 ERRCODE
      CHARACTER*300 ERRMSG

      CALL CONNECT_SYSADM(SQLCODE)
      IF (SQLCODE .NE. 0) THEN
          PRINT *, 'Failed to connect SYSADM.'

          IF (SQLCODE .NE. 0) THEN
              PRINT *, 'Failed to get diagnostics message.'
          ELSE
              PRINT *, 'errcode: ', ERRCODE
              WRITE(*, '(X,72A)') 'errmsg: ', ERRMSG
          END IF
      END IF
```

Example host application in Cobol:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 SQLCODE PIC S9(9) USAGE IS BINARY.
01 ERRCODE PIC S9(9) USAGE IS BINARY.
01 ERRMSG PICTURE X(300).

PROCEDURE DIVISION.
HEAD SECTION.

MAIN.

    CALL "CONNECT_SYSADM" USING SQLCODE.
    IF SQLCODE IS NOT ZERO
        DISPLAY "Failed to connect SYSADM."

        CALL "GET_DIAGN" USING ERRCODE, ERRMSG, SQLCODE.
        IF SQLCODE IS NOT ZERO
            DISPLAY "Failed to get diagnostics message."
        ELSE
            DISPLAY "errcode: " ERRCODE
            DISPLAY "errmsg: " ERRMSG
        END-IF.
    END-IF.
```

Example host application in Pascal:

```
type message = packed array [1..300] of char;

var
sqlcode : integer;
errcode : integer;
errmsg : message;

procedure connect_sysadm(var sqlcode : integer); external;
procedure get_diagn(var errcode : integer, var errmsg : message,
   var sqlcode : integer); external;

begin
   connect_sysadm(sqlcode);
   if sqlcode <> 0 then
   begin
      writeln('Failed to connect SYSADM.');

      get_diagn(errcode, errmsg, sqlcode);
      if sqlcode <> 0 then
      begin
         writeln('Failed to get diagnostics message.');
      end;
      else
      begin
         writeln('errcode: ', errcode)
         writeln('errmsg: ', errmsg);
      end;
```

Host Language Dependent Aspects

You can call SQL modules (MSQL) in any host language that supports the C data types generated by the MSQL preprocessor. The following host languages are supported:

- C/C++
- COBOL
- Fortran
- Pascal

Note: This is not a complete description of the rules for writing SQL modules. The programmer should use the main body of this manual as a guide to writing programs and refer to this appendix for language-specific details.

This section describes the recommended data types to use in each of the above mentioned host languages when calling an SQL module procedure.

The SQL data types `INT`(n>18), `FLOAT`, `BINARY` and `VARBINARY` are handled by Module SQL as null-terminated C char arrays, no matter the calling host language. The host application needs to apply null termination to any input, and take precautions to null termination appearing in any output.

The SQL data types `DATE`, `TIME`, `TIMESTAMP`, `INTERVAL`, `CHAR`, `VARCHAR`, `CLOB` and `BOOLEAN` are handled by Module SQL as character arrays of the given host language, see *Module declaration* on page *77* for more details.

C header files

When the LANGUAGE clause in an SQL module is set to C, a C header file containing embedded SQL statements will be generated by the MSQL preprocessor, by default with the .eh file extension. The embedded SQL header file needs to be pre-processed using the ESQL command into a pure C header file (the header file name and extension has to be specified to ESQL.) It can then be included in the C host application to provide the routine headers for the routines generated in the source code file.

Example

The SQL module currencies.msq has the LANGUAGE clause set to C. It is pre-processed using the MSQL command.

```
msql currencies.msq
```

The embedded SQL files currencies.ec and currencies.eh are generated. They are pre-processed using the ESQL commands.

```
esql --c currencies.ec
esql --header currencies.eh
```

or

```
esql /C currencies.ec
esql /HEADER currencies.eh
```

The C source file currencies.c is generated, containing the source code that calls the Mimer database. The C header file currencies.h is generated for inclusion in the C host application program.

```
#include "currencies.h"

int main
{
    int sqlcode;
    open_currencies_cursor(&sqlcode);
```

C data types

SQL data type	C data type
BIGINT	long long* int64_t*
BINARY(n)	unsigned char[n+1]
BLOB(size)	struct { long hvn_reserved; unsigned long hvn_length; char hvn_data[size]; } hvn;
BOOLEAN	char[6]
CHAR(n)	char[n+1]
CLOB(size)	struct { long hvn_reserved; unsigned long hvn_length; char hvn_data[size]; } hvn;
DATE	char[100]
DECIMAL(p) DECIMAL(p,s)	double*
DOUBLE PRECISION	double*
FLOAT	double*
FLOAT(n)	char[n+8]
INTEGER	int* int32_t*
INTEGER(n)	n <= 4: short*, int16_t* 5 <= n<= 9: int*, int32_t* 10 <= n <= 18: long long*, int64_t* 19 <= n <= 45: char[n+2]
INTERVAL data types	char[100]
NCHAR(n)	wchar_t[n+1]

SQL data type	C data type
NCLOB(size)	struct { long hvn_reserved; unsigned long hvn_length; wchar_t hvn_data[size]; } hvn;
NVARCHAR(n)	wchar_t[n+1]
REAL	float*
SMALLINT	short* int16_t*
SQLCODE	int* int32_t*
SQLSTATE	char[6]
TIME data types	char[100]
TIMESTAMP data types	char[100]
VARBINARY(n)	unsigned char[n+1]
VARCHAR(n)	char[n+1]

Note: The stdint types int16_t, int32_t and int64_t can be used in the host application if desired. The SQL module does not use them.

COBOL data types

SQL data type	COBOL data type
BIGINT	PIC S9(18) USAGE IS BINARY
BINARY(n)	PICTURE X(n+1)
BLOB(size)	01 hvn. 49 hvn-RESERVED PIC S9(18) USAGE IS BINARY. 49 hvn-LENGTH PIC S9(18) USAGE IS BINARY. 49 hvn-DATA PIC X(size).
BOOLEAN	PICTURE X(5)
CHAR(n)	PICTURE X(n)
CLOB(size)	PICTURE X(size)
DATE	PICTURE X(100)
DECIMAL(p) DECIMAL(p,0)	PIC S9(p) COMP-3
DECIMAL(p,s)	PIC S9(p)V9(s) COMP-3

SQL data type	COBOL data type
DOUBLE PRECISION	COMP-2
FLOAT	COMP-2
FLOAT(n)	PICTURE X(n+8)
INTEGER	PIC S9(9) USAGE IS BINARY
INTEGER(n)	1 <= n <= 4: PIC S9(4) USAGE IS BINARY 5 <= n <= 9: PIC S9(9) USAGE IS BINARY 10 <= n <= 18: PIC S9(18) USAGE IS BINARY 19 <= n <= 45: PICTURE X(n+2)
INTERVAL data types	PICTURE X(100)
REAL	COMP-1
SMALLINT	PIC S9(4) USAGE IS BINARY
SQLCODE	PIC S9(9) USAGE IS BINARY
SQLSTATE	PIC X(5)
TIME data types	PICTURE X(100)
TIMESTAMP data types	PICTURE X(100)
VARBINARY(n)	PICTURE X(n+1)
VARCHAR(n)	PICTURE X(n)

Fortran data types

SQL data type	Fortran data type
BIGINT	INTEGER*8
BINARY(n)	INTEGER*1(n+1)
BLOB(size)	INTEGER*1 hvn(size + 8) INTEGER*8 hvn_RESERVED INTEGER*8 hvn_LENGTH INTEGER*1 hvn_DATA(size) EQUIVALENCE(hvn(1), hvn_RESERVED) EQUIVALENCE(hvn(9), hvn_LENGTH) EQUIVALENCE(hvn(17), hvn_DATA)
BOOLEAN	CHARACTER*5
CHARACTER(n)	CHARACTER*n
CLOB(size)	CHARACTER*size
DATE	CHARACTER*100
DECIMAL(p) DECIMAL(p,s)	DOUBLE PRECISION

SQL data type	Fortran data type
DOUBLE PRECISION	DOUBLE PRECISION
FLOAT	DOUBLE PRECISION
FLOAT(n)	INTEGER*1(n+8)
INTEGER	INTEGER*4
INTEGER(n)	1 <= n <= 4: INTEGER*2 5 <= n <= 9: INTEGER*4 10 <= n <= 18: INTEGER*8 19 <= n <= 45: INTEGER*1(n+2)
INTERVAL data types	CHARACTER*100
REAL	REAL
SMALLINT	INTEGER*2
SQLCODE	INTEGER*4
SQLSTATE	CHARACTER*5
TIME data types	CHARACTER*100
TIMESTAMP data types	CHARACTER*100
VARBINARY(n)	INTEGER*1(n+1)
VARCHAR(n)	CHARACTER*n

Pascal data types

SQL data type	Pascal data type
BIGINT	integer64
BINARY(n)	packed array [1..n+1] of char

BLOB(size)	TYPE X = RECORD
	  hvn_RESERVED : INTEGER64;
	  hvn_LENGTH : INTEGER64;
	  hvn_DATA : PACKED ARRAY [1..size] OF CHAR;
	  END;

BOOLEAN	packed array [1..5] of char
CHARACTER(n)	packed array [1..n] of char
CLOB(size)	packed array [1..size] of char
DATE	packed array [1..100] of char
DECIMAL(p) DECIMAL(p,s)	double
DOUBLE PRECISION	double

SQL data type	Pascal data type
FLOAT	double
FLOAT(n)	packed array [1..n+8] of char
INTEGER	integer32
INTEGER(n)	1 <= n <= 4: integer16 5 <= n <= 9: integer32 10 <= n <= 18: integer64 19 <= n <= 45: packed array [1..n+2] of char
INTERVAL data types	packed array [1..100] of char
NCHAR(n)	packed array [1..n] of char
NCLOB(size)	TYPE X = RECORD hvn_RESERVED : INTEGER64; hvn_LENGTH : INTEGER64; hvn_DATA : PACKED ARRAY [1..size] OF CHAR; END;
NVARCHAR(n)	packed array [1..n] of char
REAL	single
SMALLINT	integer16
SQLCODE	integer32
SQLSTATE	packed array [1..5] of char
TIME data types	packed array [1..100] of char
TIMESTAMP data types	packed array [1..100] of char
VARBINARY(n)	packed array [1..n+1] of char
VARCHAR(n)	packed array [1..n] of char

Chapter 6

Mimer SQL C API

Mimer SQL C API is a native C library suitable for tool integration and application development in environments where API standardization is not a requirement. The following characteristics describe the API:

- Simplicity
- Platform independence
- Small footprint
- Tight fit with the Mimer SQL application/database communication model.

Hereinafter, the Mimer SQL C API is referred to as the Mimer API.

The Mimer SQL C API also comes in a micro environment edition under the name Mimer SQL Micro C API, which is mainly targeted for memory and CPU constrained environments. This variant is referred to as the Micro API. In most cases, the Mimer API and the Micro API routines are identical, but where there are differences, these are noted at the end of the routine description.

Architecture

The Mimer API routines may be divided into four major categories, depending on how they are used. These routine categories are:

- **Session management** - These routines manages a database session including connect, disconnect and transaction handling. See *Session Management on page 98*.
- **Statement management** - Once a session has been established, an application uses statements to interact with the database. Statements are defined using the SQL language and may be specified directly or created on the server in advance using the `CREATE STATEMENT` command. See *Statement Management on page 99*.
- **Input data management** - Used to supply input parameter data to statements. See *Data Input Routines on page 100*.
- **Output data management** - Obtains result sets and statement output parameter data. See *Data Output Routines on page 100*.

Character String Formats

API routines having string parameters come in different flavors depending on which character string format is used. The rationale is that the base routine considers all strings to be null terminated wchar_t * strings.

If a routine has string parameters, there is a companion routine suffixed with C, which accepts the string parameters as null terminated char * strings, where the character set is defined by the current locale.

Companion routines suffixed with 8 have the string format UTF-8, regardless of locale settings.

Session Management

In the Mimer API, the following routines are used for managing sessions; beginning and ending sessions, and beginning and ending transactions:

- `MimerBeginSession[C|8]`
- `MimerEndSession`
- `MimerBeginTransaction`
- `MimerEndTransaction`

The flow of calls should be according to the below. First, a session is started using a call to `MimerBeginSession` (or `MimerBeginSession8` or `MimerBeginSessionC`, depending on the data types supplied).

Then database operations take place, either separately (in auto-committed transactions), or grouped in explicit transactions. The boundaries of an explicit transaction are marked using the calls to `MimerBeginTransaction` and `MimerEndTransaction`. (See *Transactions on page 102* for more information about transactions in the Mimer API.)

This process continues until the application terminates its database session through a call to MimerEndSession.

```
MimerBeginSession[C|8]
loop
{
    MimerBeginTransaction
    <statement and data management routines>
    MimerEndTransaction
}
MimerEndSession
```

Statement Management

The routines used to manage statements are:

- MimerAddBatch
- MimerBeginStatement[C|8]
- MimerEndStatement
- MimerExecute
- MimerOpenCursor
- MimerFetch
- MimerFetchScroll
- MimerFetchSkip
- MimerCloseCursor
- MimerCurrentRow
- MimerExecuteStatement[C|8]

Which routines to use basically depends on if the statement has input or output parameters, and if a result set is returned or not.

No input or output parameters, no result set

The MimerExecuteStatement[C|8] routine is mainly intended for DDL statements (i.e. data definition language statements, e.g. create and drop table.) However, it can also be used for UPDATE, INSERT, DELETE and CALL statements without parameters.

```
MimerExecuteStatement[C|8]
```

Input or output parameters, but no result set

The MimerExecute routine is used for INSERT, UPDATE and DELETE statements, assignments (SET), and procedure calls which do not return a result set.

```
MimerBeginStatement[C|8]
<data input routines>
MimerExecute
<data output routines>
MimerEndStatement
```

Result set producing statements

Result sets are returned by `SELECT` statements, as well as by calls to result set procedures. A result set is accessed using a cursor.

```
MimerBeginStatement[C|8]
<data input routines>
MimerOpenCursor
loop
{
    MimerFetch/MimerFetchSkip/MimerFetchScroll
    <data output routines>
}
MimerCloseCursor
MimerEndStatement
```

Data Input Routines

Input data management routines are used to supply input parameter data to statements. The data management routines to set input parameter data are:

- `MimerSetBinary`
- `MimerSetBlobData`
- `MimerSetBoolean`
- `MimerSetDouble`
- `MimerSetFloat`
- `MimerSetInt32`
- `MimerSetInt64`
- `MimerSetLob`
- `MimerSetNclobData[C|8]`
- `MimerSetNull`
- `MimerSetString[C|8]`
- `MimerSetStringLen[C|8]`
- `MimerSetUUID`

Data Output Routines

The output data management routines used to obtain statement results are:

- `MimerGetBinary`
- `MimerGetBlobData`
- `MimerGetBoolean`
- `MimerGetDouble`
- `MimerGetFloat`
- `MimerGetInt32`
- `MimerGetInt64`
- `MimerGetLob`
- `MimerGetNclobData[C|8]`
- `MimerGetString[C|8]`
- `MimerGetUUID`
- `MimerIsNull`

Array Operations

The Mimer API supports the use of array fetch operations. Array fetching means that multiple rows are fetched from the server in one request. This will improve performance at the expense of memory consumption. `MimerSetArraySize` may be used to control the minimum number of rows to be fetched in each request. `MimerRowSize` may be used to determine the maximum number of bytes each row consumes. By multiplying the array size with the maximum row size a maximum array fetch memory consumption value can be obtained.

`MimerFetch` will internally fetch as many rows as possible and refill the internal buffer when needed. `MimerNext` on the other hand will return rows until all the rows in the internal buffer have been returned, it will not call the server to fetch more rows. The purpose of `MimerNext` is to have a fetch routine that is guaranteed context switch free.

The Mimer API also supports supplying parameters in arrays. Parameter arrays are specified by call `MimerAddBatch` during the parameter buildup sequence. `MimerAddBatch` adds the current set of parameters to be executed on the next `MimerExecute` call making room for another set of parameters to be specified by subsequent calls to for example `MimerSetString`.

Transactions

All operations on the database participate in a transaction. By default, operations are committed as soon as possible, which in practice means immediately after each `INSERT`, `UPDATE` or `DELETE`. A common term for this is that statements are automatically committed (autocommit).

If several operations are to be grouped together into one single transaction, the transaction boundaries must be defined by calls to `MimerBeginTransaction` and `MimerEndTransaction`.

Transactions may abort. A transaction abort is a rollback forced by the system. A transaction abort occurs when a conflict with another session has been detected, for example when two transactions have been reading and updating the same data. The database system must decide to abort one of them and to save the other one to storage. The session seeing the aborted transaction must take some action depending on this, perhaps trying to update the database once again.

Issuing large transactions puts a special burden on the database server since read- and write-sets may grow large, thus consuming memory. The read- and write-sets are logs maintaining a record of what active transactions have done. Large read- and write-sets also has the implication that they put a larger overall burden on the database server since it may take time to manage them, particularly if they grow so large that they are written to flash or magnetic memories.

Data Types

The C acronym for each SQL data type (*Mimer SQL Reference Manual, Chapter 6, Data Types in SQL Statements*), is listed in the below table.

Data type	SQL data type
MIMER_BINARY	BINARY
MIMER_BINARY_VARYING	BINARY VARYING
MIMER_BLOB	BINARY LARGE OBJECT
MIMER_BOOLEAN	BOOLEAN
MIMER_CHARACTER	CHARACTER
MIMER_CHARACTER_VARYING	CHARACTER VARYING
MIMER_CLOB	CHARACTER LARGE OBJECT
MIMER_DATE *	DATE
MIMER_DECIMAL *	DECIMAL
MIMER_FLOAT *	FLOAT(p)
MIMER_INTEGER	INTEGER(p)
MIMER_INTERVAL_DAY *	INTERVAL DAY
MIMER_INTERVAL_DAY_TO_HOUR *	INTERVAL DAY TO HOUR
MIMER_INTERVAL_DAY_TO_MINUTE *	INTERVAL DAY TO MINUTE
MIMER_INTERVAL_DAY_TO_SECOND *	INTERVAL DAY TO SECOND
MIMER_INTERVAL_HOUR *	INTERVAL HOUR
MIMER_INTERVAL_HOUR_TO_MINUTE *	INTERVAL HOUR TO MINUTE
MIMER_INTERVAL_HOUR_TO_SECOND *	INTERVAL HOUR TO SECOND
MIMER_INTERVAL_MINUTE *	INTERVAL MINUTE
MIMER_INTERVAL_MINUTE_TO_SECOND *	INTERVAL MINUTE TO SECOND
MIMER_INTERVAL_MONTH *	INTERVAL MONTH
MIMER_INTERVAL_SECOND *	INTERVAL SECOND
MIMER_INTERVAL_YEAR *	INTERVAL YEAR
MIMER_INTERVAL_YEAR_TO_MONTH *	INTERVAL YEAR TO MONTH
MIMER_NCHAR	NATIONAL CHARACTER
MIMER_NCHAR_VARYING	NATIONAL CHARACTER VARYING
MIMER_NCLOB	NATIONAL CHARACTER LARGE OBJECT

Data type	SQL data type
MIMER_T_BIGINT	BIGINT
MIMER_T_DOUBLE	DOUBLE PRECISION
MIMER_T_FLOAT	FLOAT
MIMER_T_INTEGER	INTEGER
MIMER_T_REAL	REAL
MIMER_T_SMALLINT	SMALLINT
MIMER_TIME *	TIME
MIMER_TIMESTAMP *	TIMESTAMP

The data types marked with an asterisk (*) are not supported by the Mimer API's data input and data output routines. To use these data types, convert the data using the SQL function CAST.

The following table describes which API calls may be used to set or get parameters and column values of the corresponding SQL data type.

SQL data type	Routines
BIGINT	MimerGetInt64 MimerSetInt64
BINARY	MimerGetBinary MimerSetBinary
BINARY LARGE OBJECT	MimerGetLob MimerGetBlobData MimerSetLob MimerSetBlobData
BINARY VARYING	MimerGetBinary MimerSetBinary
BOOLEAN	MimerGetBoolean MimerSetBoolean
CHARACTER	MimerGetString MimerGetString8 MimerGetStringC MimerSetString MimerSetString8 MimerSetStringC

SQL data type	Routines

CHARACTER LARGE OBJECT	`MimerGetLob`
	`MimerGetNclobData`
	`MimerGetNclobData8`
	`MimerGetNclobDataC`
	`MimerSetLob`
	`MimerSetNclobData`
	`MimerSetNclobData8`
	`MimerSetNclobDataC`

CHARACTER VARYING	`MimerGetString`
	`MimerGetString8`
	`MimerGetStringC`
	`MimerSetString`
	`MimerSetString8`
	`MimerSetStringC`

DATE	`MimerGetString`
	`MimerGetString8`
	`MimerGetStringC`
	`MimerSetString`
	`MimerSetString8`
	`MimerSetStringC`

DECIMAL	`MimerGetString`
	`MimerGetString8`
	`MimerGetStringC`
	`MimerSetString`
	`MimerSetString8`
	`MimerSetStringC`

DOUBLE PRECISION	`MimerGetDouble` `MimerSetDouble`

INTEGER	`MimerGetInt32`
	`MimerGetInt64`
	`MimerSetInt32`
	`MimerSetInt64`

NATIONAL CHARACTER	`MimerGetString`
	`MimerGetString8`
	`MimerGetStringC`
	`MimerSetString`
	`MimerSetString8`
	`MimerSetStringC`

SQL data type	Routines
NATIONAL CHARACTER LARGE OBJECT	MimerGetLob MimerGetNclobData MimerGetNclobData8 MimerGetNclobDataC MimerSetLob MimerSetNclobData MimerSetNclobData8 MimerSetNclobDataC
NATIONAL CHARACTER VARYING	MimerGetString MimerGetString8 MimerGetStringC MimerSetString MimerSetString8 MimerSetStringC
NUMERIC	MimerGetString MimerGetString8 MimerGetStringC MimerSetString MimerSetString8 MimerSetStringC
REAL	MimerGetDouble MimerGetFloat MimerSetDouble MimerSetFloat
SMALLINT	MimerGetInt32 MimerGetInt64 MimerSetInt32 MimerSetInt64
TIME	MimerGetString MimerGetString8 MimerGetStringC MimerSetString MimerSetString8 MimerSetStringC
TIMESTAMP	MimerGetString MimerGetString8 MimerGetStringC MimerSetString MimerSetString8 MimerSetStringC

Data types which do not have a corresponding manipulation method according to the above table must be explicitly casted to a data type which does.

Detecting Data Types at Run-time

If the data type of a column or parameter is not known until runtime it is possible to detect them and dynamically choose which getter or setter function to use. For this purpose the functions `MimerColumnType` and `MimerParameterType` are used to obtain the type of a column or parameter.

A range of macros may be used to determine which getter/setter which matches the data type obtained from `MimerColumnType` or `MimerParameterType`. These are:

Macro	Description
`MimerIsBinary(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetBinary` or `MimerSetBinary`.
`MimerIsBlob(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetBlobData` or `MimerSetBlobData`.
`MimerIsBoolean(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetBoolean` or `MimerSetBoolean`.
`MimerIsClob(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetNclobData[C|8]` or `MimerSetNclobData[C|8]`.
`MimerIsDouble(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetDouble` or `MimerSetDouble`.
`MimerIsFloat(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetFloat` or `MimerSetFloat`.
`MimerIsInt32(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetInt32` or `MimerSetInt32`.
`MimerIsInt64(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetInt64` or `MimerSetInt64`.
`MimerIsString(t)`	Will yield true if the type (t) can be accessed or manipulated using `MimerGetString[C|8]` or `MimerSetString[C|8]`.

Error Handling

Upon return, all routines return an integer value. In most cases the value zero (`MIMER_SUCCESS`) is used to indicates success. In some cases a positive value is used to indicate success with additional information. Negative values always indicate an error condition. The negative values are standard Mimer SQL error codes, which are listed in *Appendix B Return Codes*.

Errors between -24000 and -24299 are specific to Mimer API, where errors in the -24100 to -24199 range occur because of programming mistakes. Errors in the -24200 to -24299 range are of internal nature caused by system problems.

The Mimer API specific return codes are listed in *Mimer SQL C API Return Codes* on page *392*. The acronyms for the Mimer API specific return codes can be found in the mimerrors.h header file.

The macro `MIMER_SUCCEEDED` may be used to detect a call which has either succeeded, or succeeded with additional information (a positive value). Negating this macro may be used to detect an error.

The error condition `MIMER_SEQUENCE_ERROR` has a special meaning. It will be returned when an illegal call has been made. The Mimer API will enforce a strict sequence of allowed calls. For example, `MimerGetString` may not be called before `MimerFetch` has been called. `MIMER_SEQUENCE_ERROR` is returned when illegal sequences of calls are detected.

Error Handling Example

The below example function shows how a function uses the return value from a Mimer API function call to determine success or failure.

```
/**
 * Function which executes the statement OUR_STATEMENT
 *
 * On failure, the error code is written to stdout.
 */
#include "mimerapi.h"
int example1(MimerSession session)
{
  int32_t err, end_err=MIMER_SUCCESS;
  MimerStatement statement;

  err = MimerBeginStatement(session, L"OUR_STATEMENT", 0, &statement);
  if (MIMER_SUCCEEDED(err)) {
    err = MimerExecute(statement);
    end_err = MimerEndStatement(&statement);
  }
  if (!MIMER_SUCCEEDED(err) || !MIMER_SUCCEEDED(end_err)) {
    printf("Error %d, end error %d.\n", err, end_err);
  }
}
```

Memory Management

Throughout the Mimer API, routines are designed and implemented to allow callers not to worry about memory management. The rule is that the required buffers are allocated to their minimum size, and released as soon as they are not needed anymore.

For example, when starting a statement, information about its parameters, if it returns a result set, result set columns and their type are attached to the statement handle. This block of memory is released when the statement is released.

Memory for keeping parameters and result set columns are allocated either when the applications starts setting input parameters (see *Data Input Routines* on page *100*), or when the statement is executed (`MimerExecute`), or the cursor is opened (`MimerOpenCursor`).

When the statement is executed, the database server reads the parameters that have been set, and if there are no result set or output parameters returned, the Mimer API will release the memory promptly. If there are output parameters, or result set data, memory to keep this information is retained until the statement or cursor is closed.

Header Files Provided

When using the Mimer API, the `mimerapi.h` header file should always be included. For example, this file declares the API routines and also the handle types to be used, i.e. `MimerSession`, `MimerStatement`, `MimerLob` and `MimerHandle`. In addition, it defines a number of symbols that can be used for a convenient and instructive C coding.

Other files that are provided are the `mimerrors.h` and, on VMS only, `mimstdint.h`. Those are automatically included by the `mimerapi.h` file. The `mimstdint.h` file arranges the necessary measure for using the `int16_t`, `int32_t` and `int64_t` data types that are recommended for, and used by, the Mimer API. The `mimerrors.h` file defines C symbols for a couple of error codes that may be used when programming with the Mimer SQL C API, see *Mimer SQL C API Return Codes* on page *392*.

C symbols used in example code are defined in these header files.

Mimer API Examples

The Mimer SQL distribution contains a set of complete Mimer API based example programs; `mimerapiex1.c`, `mimerapiex2.c` and `mimerapiex3.c`.

Querying the database

The following table is used in this example:

```
create table THE_IDENT.FIRST_TABLE (
  COL1 integer primary key,
  COL2 varchar(20));
```

The example is about retrieving all rows in the table FIRST_TABLE where COL1 is larger than a supplied integer value. The rows are returned in COL1 order.

```
int err=0;
MimerSession sessionhandle;
MimerStatement statementhandle;
wchar_t res1[100];
int res2;
unsigned char res3[100];
err = MimerBeginSession(L"THE_DATABASE", L"THE_IDENT", L"THE_PASSWORD",
                        &sessionhandle); // 1
if (!err) {
  err = MimerBeginStatement(sessionhandle,
  L"select COL1, COL2 from FIRST_TABLE where COL1 > ? ORDER BY COL1",
                            MIMER_FORWARD_ONLY, &statementhandle); // 2
  if (!err) {
    err =  MimerSetInt32(statementhandle, 1, 42); // 3
    if (!err) {
      err = MimerOpenCursor(statementhandle); // 4
      if (!err) {
        do {
          err = MimerFetch(statementhandle); // 5
          if (!err) {
            MimerGetInt32(statementhandle, 1, &res2); // 6
            MimerGetString(statementhandle, 2, res1,
                           sizeof(res1)/sizeof(res1[0]));
            // You probably want to do something useful with the result here.
         }
        } while (!err);
        MimerCloseCursor(statementhandle); // 7
      }
    }
    MimerEndStatement(&statementhandle); // 8
  }
  MimerEndSession(&sessionhandle); // 9
}
```

This is an example of retrieving data from a result set. The following major events occur:

1. The `MimerBeginSession` call will start a session with the database. When a session is started we always specify which database ident (a database namespace) is the default. Above we use the THE_IDENT ident, which in practice means that we can refer to all database objects in that schema without the qualifying THE_IDENT.
2. The next thing is that we prepare the `SELECT` statement for execution. This call will load the statement into memory, along with its metadata.
3. This `SELECT` statement has one input parameter, an integer. This parameter is supplied, in this case 42.
4. A cursor is opened using `MimerOpenCursor`. We are now ready to receive data from the database.

5. Immediately when the cursor is opened, it is located before the first row (aka on row 0 of the result set). To advance the cursor position to the next row, we call `MimerFetch`.
6. Once the fetch succeeded, we may retrieve data from the row. `MimerFetch` returns a non-zero value if an error occurred or the end of the result set was reached.
7. When we have finished reading, the result set is closed.
8. If we wish to do so, we may execute the query again by calling the appropriate data input functions, `MimerOpenCursor`, `MimerFetch` etc. again. But in this example we are done and will close things down. `MimerEndStatement` is called to release the resources held by the statement.
9. Finally the database session is ended.

Retrieving a binary large object

The following table is used in this example.

```
create table PICTURES (
  ID int primary key default next value for id_sequence,
  CREATED timestamp,
  PICTURE blob);
```

In this example we have a table PICTURES with three columns; one primary key column, one column for creation timestamp and one column for a picture. We wish to return all pictures created within the last week, and with recent pictures first.

```
int err=0;
MimerSession sessionhandle;
MimerStatement statementhandle;
MimerLob blobhandle;
void *blobdata;
size_t bloblen;
int res2;
err = MimerBeginSession(L"THE_DATABASE", L"THE_IDENT", L"THE_PASSWORD",
                        &sessionhandle);
if (!err) {
  err = MimerBeginStatement(sessionhandle,
L"select CREATED, PICTURE from PICTURES where CREATED >= localtimestamp -
interval'7' days order by CREATED desc",
                            MIMER_FORWARD_ONLY, &statementhandle);
  if (!err) {
    err = MimerOpenCursor(statementhandle);
    if (!err) {
      while (!err) {
        err = MimerFetch(statementhandle);
        if (!err) {
          MimerGetInt32(statementhandle, 1, &res2);
          MimerGetLob(statementhandle, 2, &bloblen, &blobhandle);
          blobdata = malloc(bloblen);
          if (blobdata) {
            err = MimerGetBlobData(&blobhandle, blobdata, bloblen);
            if (!err) {
            // You probably want to do something useful with the blob here
            }
            free(blobdata);
          }
        }
      }
      MimerCloseCursor(statementhandle);
    }
    MimerEndStatement(&statementhandle);
  }
  MimerEndSession(&sessionhandle);
}
```

Inserting a binary large object into the database

The following table is used in this example.

```
create table THIRD_TABLE (
  COL1 integer primary key,
  COL2 blob);
```

This example features binary large objects which are numbered. We have created a new object whose identity number is 42411 that we want to insert into the database.

```
int err=0;
MimerSession sessionhandle;
MimerStatement statementhandle;
MimerLob blobhandle;
char *blobdata;
int bloblen;
int param1 = 42411;
unsigned char res3[100];
// Below, the location of the binary large object data is obtained.
blobdata = _some_interesting_location_; // 1
bloblen = 47110;
err = MimerBeginSession(L"THE_DATABASE", L"THE_IDENT", L"THE_PASSWORD",
                        &sessionhandle);
if (!err) {
  err = MimerBeginStatement(sessionhandle,
            L"insert into THIRD_TABLE (COL1, COL2) values (?, ?)",
            MIMER_FORWARD_ONLY, &statementhandle);
  if (!err) {
     MimerSetInt32(statementhandle, 1, param1);
    err = MimerSetLob(statementhandle, 2, bloblen, &blobhandle); // 2
    if (!err) {
      err = MimerSetBlobData(&blobhandle, &blobdata[0], 10000); // 3
      err = MimerSetBlobData(&blobhandle, &blobdata[10000], 10000);
      err = MimerSetBlobData(&blobhandle, &blobdata[20000], 10000);
      err = MimerSetBlobData(&blobhandle, &blobdata[30000], 10000);
      err = MimerSetBlobData(&blobhandle, &blobdata[40000], 7110); // 4
      err = MimerExecute(statementhandle); // 5
    }
    MimerEndStatement(&statementhandle);
  }
  MimerEndSession(&sessionhandle);
}
```

The following interesting things takes place in this example:

1. In some way we obtain the location and length of the data to store in the database.
2. The process of storing the blob is started. The total size of the blob is supplied.
3. In this case the object data is supplied in chunks of 10 000 bytes. `MimerSetBlobData` is therefore called five times. Error handling is omitted here for clarity.

   Choosing the chunk size is a compromise between memory consumption and performance. If it is important to reduce memory usage, it may be appropriate to process the object sequentially in smaller pieces, rather than to read everything into memory.
4. The final call to `MimerSetBlobData` supplies the remaining 7110 bytes.
5. The actual INSERT operation is performed.

Scrolling through a result set

The following table is used in this example:

```
create table THE_IDENT.FIRST_TABLE (
  COL1 integer primary key,
  COL2 varchar(20));
```

This example is basically the same as in *Querying the database* on page *111*, except that we are performing some scrolling operations on the result set. The number of rows in the result set in this example is 10.

Retrieve all rows in the table FIRST_TABLE whose primary key (an integer) is larger than a supplied value, and scroll through the result set.

```
int err=0;
MimerSession sessionhandle;
MimerStatement statementhandle;
int current_row;
err = MimerBeginSession(L"THE_DATABASE", L"THE_IDENT", L"THE_PASSWORD",
                        &sessionhandle);
if (!err) {
  err = MimerBeginStatement(sessionhandle,
  L"select COL1, COL2 from FIRST_TABLE where COL1 > ? ORDER BY COL1",
                            MIMER_SCROLLABLE, &statementhandle); // 2
  if (!err) {
    err =  MimerSetInt32(statementhandle, 1, 42);
    if (!err) {
      err = MimerOpenCursor(statementhandle);
      if (!err) {
        current_row = MimerCurrentRow(statementhandle); // current_row=0
        do {
          err = MimerFetchScroll(statementhandle, MIMER_NEXT, 0); // 1
          current_row = MimerCurrentRow(statementhandle); // current_row=1
          [...]
          err = MimerFetchScroll(statementhandle, MIMER_RELATIVE, -3); // 2
          current_row = MimerCurrentRow(statementhandle); // current_row=0
          [...]
          err = MimerFetchScroll(statementhandle, MIMER_ABSOLUTE, 10); // 3
          current_row = MimerCurrentRow(statementhandle); // current_row=10
          [...]
          err = MimerFetchScroll(statementhandle, MIMER_PREVIOUS, 0); // 4
          current_row = MimerCurrentRow(statementhandle); // current_row=9
          [...]
          err = MimerFetchScroll(statementhandle, MIMER_ABSOLUTE, 20); // 5
          current_row = MimerCurrentRow(statementhandle); // current_row=11
          [...]
          err = MimerFetchScroll(statementhandle, MIMER_LAST, 0); // 6
          current_row = MimerCurrentRow(statementhandle); // current_row=10
          [...]
          err = MimerFetchScroll(statementhandle, MIMER_FIRST, 0); // 7
          current_row = MimerCurrentRow(statementhandle); // current_row=1
          [...]
        } while (!err);
        MimerCloseCursor(statementhandle);
      }
    }
    MimerEndStatement(&statementhandle);
  }
  MimerEndSession(&sessionhandle);
}
```

1 We scroll one row forward, into the first row of the result set. The current row is now 1.

2 We now scroll three rows backwards. The current row is now before the result set. Even though we scroll three rows backwards, we cannot get further back than the row before the result set. The fetch call will return `MIMER_NO_DATA` and the current row is 0.

3 Now, we scroll to the tenth row. The current row is now 10.

4 One row backwards. The current row is now 9.

5 We try to scroll to the twentieth row. Since there are only 10 rows in the result set, the scroll operation will return `MIMER_NO_DATA`, and the current row is now 11.

6 We wish to see the last row. The current row is now 10.

7 Now, we scroll back to the first row. The current row is now 1.

Chapter 7

Mimer SQL C API Reference

The following routines are included in the Mimer API:

Routine	Description	Micro API compatible
`MimerAddBatch`	Add currently set parameters to statement.	Yes
`MimerBeginSession`	Starts a session with the database. (wchar_t version.)	Yes
`MimerBeginSession8`	Starts a session with the database. (UTF-8 version.)	
`MimerBeginSessionC`	Starts a session with the database. (char version.)	Yes
`MimerBeginStatement`	Prepares a statement for execution. (wchar_t version.)	Yes
`MimerBeginStatement8`	Prepares a statement for execution. (UTF-8 version.)	
`MimerBeginStatementC`	Prepares a statement for execution. (char version.)	Yes
`MimerBeginTransaction`	Starts a transaction.	
`MimerCloseCursor`	Closes an open cursor.	Yes
`MimerColumnCount`	Obtains the number of columns in a result set.	
`MimerColumnName`	Obtains the name of a column. (wchar_t version.)	
`MimerColumnName8`	Obtains the name of a column. (UTF-8 version.)	
`MimerColumnNameC`	Obtains the name of a column. (char version.)	

Routine	Description	Micro API compatible
MimerColumnType	Returns the type of a column.	
MimerCurrentRow	Returns the current row of a result set.	
MimerEndSession	Ends a database session.	Yes
MimerEndStatement	Closes a prepared statement.	Yes
MimerEndTransaction	Commits or rollbacks a transaction.	
MimerExecute	Executes a statement that does not return a result set.	Yes
MimerExecuteStatement	Executes a statement directly without parameters. (wchar_t version.)	Yes
MimerExecuteStatement8	Executes a statement directly without parameters. (UTF-8 version.)	
MimerExecuteStatementC	Executes a statement directly without parameters. (char version.)	Yes
MimerFetch	Advances to the next row of the result set.	Yes
MimerFetchScroll	Moves the current cursor position on a scrollable cursor.	
MimerFetchSkip	Advances to the next row of the result set but optionally skips a number of rows in the result set.	Yes
MimerGetBinary	Gets binary data from a result set or an output parameter.	Yes
MimerGetBlobData	Retrieves the content of a binary large object.	
MimerGetBoolean	Gets boolean data from a result set or an output parameter.	Yes
MimerGetDouble	Gets double precision float data from a result set or an output parameter.	Yes
MimerGetFloat	Gets single precision float data from a result set or an output parameter.	Yes
MimerGetInt32	Gets int32_t integer data from a result set or an output parameter.	Yes
MimerGetInt64	Gets int64_t integer (long long) data from a result set or an output parameter.	Yes
MimerGetLob	Obtains a large object handle.	

Routine	Description	Micro API compatible
MimerGetNclobData	Retrieves the contents of a character large object. (wchar_t version.)	
MimerGetNclobData8	Retrieves the contents of a character large object. (UTF-8 version.)	
MimerGetNclobDataC	Retrieves the contents of a character large object. (char version.)	
MimerGetStatistics	Obtains server statistics information.	Yes
MimerGetString	Gets character data from a result set or an output parameter. (wchar_t version.)	Yes
MimerGetString8	Gets character data from a result set or an output parameter. (UTF-8 version.)	
MimerGetStringC	Gets character data from a result set or an output parameter into a multi byte character string. (char version.)	Yes
MimerGetUUID	Gets a Universally unique identifier from a result set or output parameter.	Yes
MimerIsNull	Checks if a result set column or output parameter has the SQL null value.	Yes
MimerNext	Advances the current row to the next row, within the current array.	
MimerOpenCursor	Opens a result set.	Yes
MimerParameterCount	Returns the number of parameters for a statement.	
MimerParameterMode	Detects the input/output mode of a parameter.	
MimerParameterName	Obtains the name of a parameter. (wchar_t version.)	
MimerParameterName8	Obtains the name of a parameter. (UTF-8 version.)	
MimerParameterNameC	Obtains the name of a parameter. (char version.)	
MimerParameterType	Obtains the data type of a parameter.	
MimerSetArraySize	Sets the number of bytes to fetch in each server request.	
MimerSetBinary	Sets a binary data parameter.	Yes
MimerSetBlobData	Sets the data of a binary large object.	

Routine	Description	Micro API compatible
MimerSetBoolean	Sets a boolean data parameter.	Yes
MimerSetDouble	Sets a double precision floating point parameter.	Yes
MimerSetFloat	Sets a single precision floating point parameter.	Yes
MimerSetInt32	Sets an int32_t integer parameter.	Yes
MimerSetInt64	Sets an int64_t integer parameter.	Yes
MimerSetLob	Sets a large object in the database.	
MimerSetNclobData	Sets the data of a character large object. (wchar_t version.)	
MimerSetNclobData8	Sets the data of a character large object. (UTF-8 version.)	
MimerSetNclobDataC	Sets the data of a character large object. (char version.)	
MimerSetNull	Sets an input parameter to the SQL null value.	Yes
MimerSetString	Sets a string parameter. (wchar_t version.)	Yes
MimerSetString8	Sets a string parameter. (UTF-8 version.)	
MimerSetStringC	Sets a string parameter. (char version.)	Yes
MimerSetStringLen	Sets a string parameter. (wchar_t version.)	
MimerSetStringLen8	Sets a string parameter. (UTF-8 version.)	
MimerSetStringLenC	Sets a string parameter. (char version.)	
MimerSetUUID	Sets a Universally unique identifier parameter.	Yes

MimerAddBatch

Add the currently set parameters to the statement be executed on the next call to `MimerExecute`. If this call succeeds, the statement is ready to accept another set of parameters to be executed during the same server call. The benefit of executing several parameter sets at the same time is beneficial to performance.

Statements which may accept multiple parameter sets include DML statement and procedure calls, such as INSERT, UPDATE, DELETE and CALL. `MimerAddBatch` cannot be used with statements which return result sets.

Parameters

```
int32_t MimerAddBatch (
            MimerStatement statementhandle)
```

`statementhandle`	in	The statement whose parameters to batch.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The statementhandle parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	The statement was not ready to accept parameters.
`MIMER_UNSET_PARAMETERS`	All input parameters have not yet been set.
`MIMER_OUT_OF_MEMORY`	Enough memory to hold an additional set of parameters could not be allocated.

Notes

This routine may interact with the database server.

Micro API compatible.

MimerBeginSession

Starts a session with the database. (wchar_t version.)

Parameters

```
int32_t MimerBeginSession (
            const wchar_t *database,
            const wchar_t *ident,
            const wchar_t *password,
            MimerSession *sessionhandle)
```

database	in	The name of the database to connect to. If this is null, a connection to the default database is created.
ident	in	The ident associated with the session to create. This parameter may not be null.
password	in	The password of the ident. A null value is identical to a zero length password. Database servers which do not enforce authentication control may ignore the password.
sessionhandle	out	A session handle identifying the session when calling MimerEndSession, MimerBeginStatement, MimerBeginStatement8, MimerBeginStatementC, MimerBeginTransaction, MimerExecuteStatement, MimerExecuteStatement8, MimerExecuteStatementC and MimerEndTransaction.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
MIMER_SUCCESS	Success.
MIMER_OUTOFMEMORY	Out of memory.
MIMER_ILLEGAL_CHARACTER	One of the string parameters database, ident or password contained an illegal character.
< 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

A session handle may or may not have been returned. MimerEndSession should therefore always be called to avoid handle leaks.

Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerBeginSession8

Starts a session with the database. (UTF-8 version.)

Parameters

```
int32_t MimerBeginSession8 (
            const char *database,
            const char *ident,
            const char *password,
            MimerSession *sessionhandle)
```

`database`	in	The name of the database to connect to. If this is null, a connection to the default database is created.
`ident`	in	The ident associated with the session to create. This parameter may not be null.
`password`	in	The password of the ident. A null value is identical to a zero length password. Database servers which do not enforce authentication control may ignore the password.
`sessionhandle`	out	A session handle identifying the session when calling `MimerEndSession`, `MimerBeginStatement`, `MimerBeginStatement8`, `MimerBeginStatementC`, `MimerBeginTransaction`, `MimerExecuteStatement`, `MimerExecuteStatement8`, `MimerExecuteStatementC` and `MimerEndTransaction`.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_OUTOFMEMORY`	Out of memory.
`MIMER_ILLEGAL_CHARACTER`	One of the string parameters database, ident or password contained an illegal character.
< 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

A session handle may or may not have been returned. `MimerEndSession` should therefore always be called to avoid handle leaks.

Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerBeginSessionC

Starts a session with the database. (char version.)

Parameters

```
int32_t MimerBeginSessionC (
            const char *database,
            const char *ident,
            const char *password,
            MimerSession *sessionhandle)
```

database	in	The name of the database to connect to. If this is null, a connection to the default database is created.
ident	in	The ident associated with the session to create. This parameter may not be null.
password	in	The password of the ident. A null value is identical to a zero length password. Database servers which do not enforce authentication control may ignore the password.
sessionhandle	out	A session handle identifying the session when calling MimerEndSession, MimerBeginStatement, MimerBeginStatement8, MimerBeginTransaction, MimerExecuteStatement, MimerExecuteStatement8, MimerExecuteStatementC and MimerEndTransaction.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
MIMER_SUCCESS	Success.
MIMER_OUTOFMEMORY	Out of memory.
MIMER_ILLEGAL_CHARACTER	One of the string parameters database, ident or password contained an illegal character.
< 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

A session handle may or may not have been returned. MimerEndSession should therefore always be called to avoid handle leaks.

Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerBeginStatement

Prepares an SQL statement for execution. (wchar_t version.)

Between the preparation and the time of execution, the application may supply any number of input parameters to be used when executing the statement.

The same statement does not need to be prepared again, even if it is executed multiple times. However, all input parameters have to be set again before each execution.

There is no need to prepare all statements at the start of the application.

Parameters

```
int32_t MimerBeginStatement (
            MimerSession sessionhandle,
            const wchar_t *sqlstatement,
            int32_t options,
            MimerStatement *statementhandle)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C]`, identifying the session.
`sqlstatement`	in	SQL statement string.
`options`	in	A bit mask of options identifying the characteristics of the statement. The following values specifies a cursor being scrollable, or forward only: `MIMER_FORWARD_ONLY` (`0x0`) `MIMER_SCROLLABLE` (`0x1`) A value of `0` will indicate `MIMER_FORWARD_ONLY`.
`statementhandle`	out	A handle to the statement is returned here.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_ILLEGAL_CHARACTER`	The statement string contained an illegal character.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_STATEMENT_CANNOT_BE_PREPARED`	The statement was a DDL statement which cannot be prepared. Such statements should be executed using `MimerExecuteStatement`.
`MIMER_TRUNCATION_ERROR`	The statement string was too long.

Return value	Description
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerBeginStatement8

Prepares an SQL statement for execution. (UTF-8 version.)

Between the preparation and the time of execution, the application may supply any number of parameters to be used when executing the statement.

The same statement does not need to be prepared again, even if it is executed multiple times.

There is no need to prepare all statements at the start of the application.

Parameters

```
int32_t MimerBeginStatement8 (
            MimerSession sessionhandle,
            const char *sqlstatement,
            int32_t options,
            MimerStatement *statementhandle)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C]`, identifying the session.
`sqlstatement`	in	SQL statement string.
`options`	in	A bit mask of options identifying the characteristics of the statement. The following values specifies a cursor being scrollable, or forward only: `MIMER_FORWARD_ONLY` (`0x0`) `MIMER_SCROLLABLE` (`0x1`) A value of `0` will indicate `MIMER_FORWARD_ONLY`.
`statementhandle`	out	A handle to the statement is returned here.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_ILLEGAL_CHARACTER`	The statement string contained an illegal character.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_STATEMENT_CANNOT_BE_PREPARED`	The statement was a DDL statement which cannot be prepared. Such statements should be executed using `MimerExecuteStatement`.
`MIMER_TRUNCATION_ERROR`	The statement string was too long.

Return value	Description
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerBeginStatementC

Prepares an SQL statement for execution. (char version.)

Between the preparation and the time of execution, the application may supply any number of parameters to be used when executing the statement.

The same statement does not need to be prepared again, even if it is executed multiple times.

There is no need to prepare all statements at the start of the application.

Parameters

```
int32_t MimerBeginStatementC (
            MimerSession sessionhandle,
            const char *sqlstatement,
            int32_t options,
            MimerStatement *statementhandle)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C]`, identifying the session.
`sqlstatement`	in	SQL statement string.
`options`	in	A bit mask of options identifying the characteristics of the statement. The following values specifies a cursor being scrollable, or forward only: `MIMER_FORWARD_ONLY` (`0x0`) `MIMER_SCROLLABLE` (`0x1`) A value of `0` will indicate `MIMER_FORWARD_ONLY`.
`statementhandle`	out	A handle to the statement is returned here.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_ILLEGAL_CHARACTER`	The statement string contained an illegal character.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_STATEMENT_CANNOT_BE_PREPARED`	The statement was a DDL statement which cannot be prepared. Such statements should be executed using `MimerExecuteStatement`.
`MIMER_TRUNCATION_ERROR`	The statement string was too long.

Return value	Description
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerBeginTransaction

Starts a transaction.

This routine only needs to be called if two or more database operations should participate in the transaction. (If the transaction consists of one single operation, simply use the auto-commit functionality, which is the default. See *Transactions* on page *102*.)

Parameters

```
int32_t MimerBeginTransaction (
            MimerSession sessionhandle,
            int32_t transoption)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C|8]`, identifying the session.
`transoption`	in	A bit mask of options identifying the characteristics of the transaction. The following values specifies a transaction being read/write, or read only: `MIMER_TRANS_READWRITE` (`0x0`) `MIMER_TRANS_READONLY` (`0x4`) A value of `0` will indicate `MIMER_TRANS_READWRITE`.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

MimerCloseCursor

Closes an open cursor.

When the cursor is closed, all resources held by the result set are released.

After the cursor is closed, the cursor may either be reopened using a call to `MimerOpenCursor` or the statement should be released using a call to `MimerEndStatement`. Before the cursor is reopened, a new set of parameters may be supplied using any of the data input functions.

Parameters

```
int32_t MimerCloseCursor (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

MimerColumnCount

Obtains the number of columns in a result set.

Parameters

```
int32_t MimerColumnCount (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.

Returns

If zero or positive, the number of result set columns. If negative a standard Mimer error code.

Return value	Description
>= 0	The number of result set columns.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	The statement is not compiled.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

MimerColumnName

Obtains the name of a column. (wchar_t version.)

Parameters

```
int32_t MimerColumnName (
            MimerStatement statementhandle,
            int16_t column,
            wchar_t *columnname,
            size_t maxlen)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`column`	in	The column number, where the leftmost column is 1.
`columnname`	out	The area where to return the column name. The maximum column name length returned is `maxlen-1`.
`maxlen`	in	The size of the `columnname` area.

Returns

Returns the number of characters in the column name. If this value is larger than `maxlen-1`, a truncation occurred. If a negative value was returned, there was an error.

Return value	Description
>= 0	Length of column name (in characters)
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	Statement is not compiled
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The column number does not exist
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerColumnName8

Obtains the name of a column. (UTF-8 version.)

Parameters

```
int32_t MimerColumnName8 (
            MimerStatement statementhandle,
            int16_t column,
            char *columnname,
            size_t maxsiz)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`column`	in	The column number, where the leftmost column is 1.
`columnname`	out	The are where to return the column name. The maximum column name length returned is `maxsiz-1`.
`maxsiz`	in	The size of the `columnname` area.

Returns

Returns the size of the column name in bytes. If this value is larger than `maxsiz-1`, a truncation occurred. If a negative value was returned, there was an error.

Return value	Description
>= 0	Length of column name (in characters)
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	Statement is not compiled.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The column number does not exist.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerColumnNameC

Obtains the name of a column. (char version.)

Parameters

```
int32_t MimerColumnNameC (
            MimerStatement statementhandle,
            int16_t column,
            char *columnname,
            size_t maxsiz)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`column`	in	The column number, where the leftmost column is 1.
`columnname`	out	The are where to return the column name. The maximum column name length returned is `maxsiz-1`.
`maxsiz`	in	The size of the `columnname` area.

Returns

Returns the size of the column name in bytes. If this value is larger than `maxsiz-1`, a truncation occurred. If a negative value was returned, there was an error.

Return value	Description
>= 0	Length of column name (in characters)
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	Statement is not compiled.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The column number does not exist.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerColumnType

Returns the type of a column.

Parameters

```
int32_t MimerColumnType (
            MimerStatement statementhandle,
            int16_t column)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`column`	in	The column number, where the leftmost column is 1.

Returns

Returns the column type or a negative value if an error occurred.

Return value	Description
> 0	Data type. See list in *Data Types* on page *103*.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	Statement is not compiled
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The column number does not exist
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

MimerCurrentRow

Returns the current row of a result set.

Parameters

```
int32_t MimerCurrentRow (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.

Returns

A negative value indicate an error code. Zero or a positive value indicates the current cursor position.

Return value	Description
>= 0	Current cursor position. If the current position is before the result set, 0 is returned. A value of 1 indicates the first row of the result set.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open.

Notes

Not Micro API compatible.

MimerEndSession

Ends a database session.

If there are any active transactions, these are rollbacked.

Parameters

```
int32_t MimerEndSession (MimerSession *sessionhandle)
```

`sessionhandle`	in	A reference to a handle returned by `MimerBeginSession[C|8]`, identifying the session.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	If there are open statements on the session. Use `MimerEndStatement` to release them.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

MimerEndStatement

Closes a prepared statement.

If there is an open cursor on this statement, it is automatically closed.

Parameters

```
int32_t MimerEndStatement (MimerStatement *statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

MimerEndTransaction

Commits or rollbacks a transaction.

Open cursors are automatically closed.

Parameters

```
int32_t MimerEndTransaction (
            MimerSession sessionhandle,
            int32_t commit_rollback)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C|8]` identifying the session.
`commit_rollback`	in	An integer specifying the transaction operation to perform. `MIMER_COMMIT` and `MIMER_ROLLBACK` are recognized.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	No transaction has been started.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

MimerExecute

Execute a statement that does not return a result set.

Statements that do not return a result set include data manipulation statements (INSERT, UPDATE, DELETE), assignments (SET) and procedure calls (CALL). (Statements that return a result set are handled using cursors. See *Result set producing statements* on page *100*.)

Before calling `MimerExecute`, set all input parameters of the statement using data input routines. (See *Data Input Routines* on page *100*.)

If this call returns successfully, any output parameters may be retrieved using data output routines. (See *Data Output Routines* on page *100*.)

Parameters

```
int32_t MimerExecute (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.

Returns

A negative value indicating an error, or zero if successful. A positive value indicates an update row count returned by the database.

Return value	Description
> 0	Success. Value indicates an update row count.
`MIMER_SUCCESS`	Success.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_SEQUENCE_ERROR`	If this routine is called on a statement which returns a result set.
`MIMER_UNSET_PARAMETERS`	All input parameters have not yet been set.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

MimerExecuteStatement

Executes a statement directly without parameters. (wchar_t version.)

This routine is mainly intended for data definition statements (e.g. CREATE TABLE), but can also be used for regular INSERT, UPDATE and DELETE statements with no input or output parameters.

Parameters

```
int32_t MimerExecuteStatement (
            MimerSession sessionhandle,
            const wchar_t *sqlstatement)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C|8]`, identifying the session.
`sqlstatement`	in	An SQL statement.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_ILLEGAL_CHARACTER`	The statement string contained an illegal character.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_STRING_TRUNCATION`	The statement string was too long.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerExecuteStatement8

Executes a statement directly without parameters. (UTF-8 version.)

This routine is mainly intended for data definition statements (e.g. CREATE TABLE), but can also be used for regular INSERT, UPDATE and DELETE statements with no input or output parameters.

Parameters

```
int32_t MimerExecuteStatement8 (
            MimerSession sessionhandle,
            const char *sqlstatement)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C|8]`, identifying the session.
`sqlstatement`	in	An SQL statement.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_ILLEGAL_CHARACTER`	The statement string contained an illegal character.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_STRING_TRUNCATION`	The statement string was too long.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerExecuteStatementC

Executes a statement directly without parameters. (char version.)

This routine is mainly intended for data definition statements (e.g. CREATE TABLE), but can also be used for regular INSERT, UPDATE and DELETE statements with no input or output parameters.

Parameters

```
int32_t MimerExecuteStatementC (
            MimerSession sessionhandle,
            const char *sqlstatement)
```

`sessionhandle`	in	A handle returned by `MimerBeginSession[C|8]`, identifying the session.
`sqlstatement`	in	An SQL statement.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_ILLEGAL_CHARACTER`	The statement string contained an illegal character.
`MIMER_HANDLE_INVALID`	The `sessionhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_STRING_TRUNCATION`	The statement string was too long.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerFetch

Advances to the next row of the result set.

If the routine returns successfully, the current row has been advanced one row down the result set. Column data may be retrieved using data output routines.

Parameters

```
int32_t MimerFetch (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement with an open cursor.

Returns

A negative value indicating an error, or zero if successful. A value of `MIMER_NO_DATA` indicates that the end of the result has been reached.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	If there is no open result set on this statement.
`MIMER_NO_DATA`	If there are no more rows in the result set.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

MimerFetchScroll

Moves the current cursor position on a scrollable cursor.

A non-scrollable cursor (`MIMER_FORWARD_ONLY`) may only be read using scroll operation `MIMER_NEXT`.

Parameters

```
int32_t MimerFetchScroll (
            MimerStatement statementhandle,
            int32_t operation,
            int32_t offset)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`operation`	in	A value describing the scroll operation to perform. This value may be either of the following: `MIMER_RELATIVE` (`0x00000000`) - Move to a row number relative to the current position. `MIMER_NEXT` (`0x00000001`) - Move on to the next row. `MIMER_PREVIOUS` (`0xffffffff`) - Move to the previous row. `MIMER_ABSOLUTE` (`0x40000000`) - Move to an absolute row number. `MIMER_FIRST` (`0x40000001`) - Move to the first row of the result set. `MIMER_LAST` (`0xbfffffff`) - Move to the last row of the result set.
`offset`	in	A parameter to `MIMER_RELATIVE` and `MIMER_ABSOLUTE` operation codes. A relative value of *n* will move the cursor *n* rows relative to the current row. An absolute value of *n* will move to the *n*:th row of the result set. See *Scrolling through a result set on page 114* for an example of scroll operations.

Returns

A negative value indicating an error, or zero if successful. A value of `MIMER_NO_DATA` indicates that the end of the result has been reached.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.

Return value	Description
MIMER_SEQUENCE_ERROR	There is no open scrollable cursor this statement. Scrollability is determined at compile time with the options parameter to MimerBeginStatement or MimerBeginStatement8.
MIMER_NO_DATA	If the scroll operation ended up on a row without data, that is either before or after the result set.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

MimerFetchSkip

Advances to the next row of the result set but optionally skips a number of rows in the result set.

If the routine returns successfully, the current row has been advanced the specified number of rows down the result set. Column data may be retrieved using any of the data output functions.

Parameters

```
int32_t MimerFetchSkip (
            MimerStatement statementhandle,
            int32_t count)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement with an open cursor.
`count`	in	Number of rows to advance the current row down the result set.

Returns

A negative value indicating an error, or zero if successful. A value of `MIMER_NO_DATA` indicates that the end of the result has been reached.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NO_DATA`	If there are no more rows in the result set.
`MIMER_SEQUENCE_ERROR`	If there is no open result set on this statement.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

MimerGetBinary

Gets binary data from a result set or an output parameter.

Only SQL data types BINARY and BINARY VARYING may be retrieved using this routine.

Parameters

```
int32_t MimerGetBinary (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            void *dest,
            size_t dest_maxsiz)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`dest`	out	A memory location where to place output binary data. If this is null, no data will be returned, but the size of the binary data is still returned by this routine.
`dest_maxsiz`	in	The maximum size of the `dest` memory location in bytes.

Returns

A negative value indicating error, or a non-negative value indicating the number of bytes in the binary column.

Return value	Description
>= 0	The number of bytes in the binary column. If this value is larger than the `dest_maxsiz` parameter, the output data was truncated.
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the BINARY or BINARY VARYING data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.

Return value	Description
`MIMER_SQL_NULL_VALUE`	The column or output parameter contained the SQL null value. (This can also be detected using the `MimerIsNull` routine.)

Notes

Micro API compatible.

MimerGetBlobData

Retrieves the content of a binary large object.

If the binary large object is to be retrieved in multiple chunks, chunks are retrieved in sequence, where the length of each chunk is specified by the length parameter of each call to this routine. Chunks may only be retrieved sequentially through the binary large object.

Parameters

```
int32_t MimerGetBlobData (
            MimerLob *blobhandle,
            void *data,
            size_t size)
```

`blobhandle`	in	A reference to a binary large object handle created by `MimerGetLob`.
`data`	out	Where to place the entire or a part of the binary large object.
`size`	in	The size in bytes to retrieve in this chunk.

Returns

A negative value indicating error, or a non-negative value indicating the number of bytes available to return.

Return value	Description
>= 0	The number of bytes available to return. This includes the number of bytes returned in this call and all data that are left in the BLOB. When the returned value is equal or less than the size input parameter, all data of the BLOB has been returned.
`MIMER_HANDLE_INVALID`	The `blobhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

MimerGetBoolean

Gets boolean data from a result set or an output parameter.

Only the database data type BOOLEAN may be retrieved using this call.

Parameters

```
int32_t MimerGetBoolean (
            MimerStatement statementhandle,
            int16_t paramno_colno)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.

Returns

The value 0 for FALSE, the value 1 for TRUE, or a negative value indicating an error.

Return value	Description
`1`	Boolean value TRUE.
`0`	Boolean value FALSE.
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the BOOLEAN data type.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
`MIMER_SQL_NULL_VALUE`	The column or output parameter contained the SQL null value. (This can also be detected using the `MimerIsNull` routine.)

Notes

Micro API compatible.

MimerGetDouble

Gets double precision floating point data from a result set or an output parameter.

Only the SQL data types DOUBLE PRECISION and REAL may be retrieved using this call.

Parameters

```
int32_t MimerGetDouble (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            double *value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`value`	out	A memory location where to place the double precision floating point value.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the output value was not of the DOUBLE PRECISION or REAL data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
`MIMER_SQL_NULL_VALUE`	The column or output parameter contained the SQL null value. (This can also be detected using the `MimerIsNull` routine.)

Notes

Not Micro API compatible.

MimerGetFloat

Gets single precision floating point data from a result set or an output parameter.

Only the SQL data types DOUBLE PRECISION and REAL may be retrieved using this call.

Parameters

```
int32_t MimerGetFloat (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            float *value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`value`	out	A memory location where to place the floating point value.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the output value was not of the DOUBLE PRECISION or REAL data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
`MIMER_SQL_NULL_VALUE`	The column or output parameter contained the SQL null value. (This can also be detected using the `MimerIsNull` routine.)

Notes

Micro API compatible.

MimerGetInt32

Gets integer data from a result set or an output parameter.

Only the SQL data types INTEGER, BIGINT and SMALLINT may be get using this call.

Parameters

```
int32_t MimerGetInt32 (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            int32_t *value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`value`	out	A memory location where to place the integer.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the INTEGER, BIGINT or SMALLINT data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
`MIMER_SQL_NULL_VALUE`	The column or output parameter contained the SQL null value. (This can also be detected using the `MimerIsNull` routine.)

Notes

Micro API compatible.

MimerGetInt64

Gets int64_t integer data from a result set or an output parameter.

Only the SQL data types INTEGER, BIGINT and SMALLINT may be retrieved using this call.

Parameters

```
int32_t MimerGetInt64 (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            int64_t *value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`value`	out	A memory location where to place the integer value.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the value was not of the INTEGER, BIGINT or SMALLINT data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
`MIMER_SQL_NULL_VALUE`	The column or output parameter contained the SQL null value. (This can also be detected using the `MimerIsNull` routine.)

Notes

Micro API compatible.

MimerGetLob

Obtains a large object handle.

Only the SQL data types BINARY LARGE OBJECT, CHARACTER LARGE OBJECT and NATIONAL CHARACTER LARGE OBJECT may be retrieved using this call.

Whenever a large object is about to retrieved from the database, this routine obtains the object from the result set row, and the length is returned. Subsequent calls to `MimerGetBlobData` or `MimerGetNclobData` retrieves the actual object from the database.

If there are any other open large object handles on this statement, these are closed before returning the new one.

Note that if this routine returns a size or length of 0, it is strictly not required to call the `MimerGetBlobData` or `MimerGetNclobData` routines for the large object contents, thus a database server round-trip may be saved.

Parameters

```
int32_t MimerGetLob (
            MimerStatement statementhandle,
            int16_t colno,
            size_t *size_length,
            MimerLob *lobhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`colno`	in	The column number to get data from. First column is 1.
`size_length`	out	The size of a BINARY LARGE OBJECT in bytes or the length of a CHARACTER LARGE OBJECT or a NATIONAL CHARACTER LARGE OBJECT.
`lobhandle`	out	A handle to the binary large object.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the BINARY LARGE OBJECT, CHARACTER LARGE OBJECT or NATIONAL CHARACTER LARGE OBJECT data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.

Return value	Description
MIMER_PARAMETER_NOT_OUTPUT	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
MIMER_SEQUENCE_ERROR	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
MIMER_SQL_NULL_VALUE	The column or output parameter contained the SQL null value. (This can also be detected using the MimerIsNull routine.)

Notes

Not Micro API compatible.

MimerGetNclobData

Retrieves the contents of a character large object. (wchar_t version.)

A character large object may be retrieved in one or more chunks.

If the character large object is to be retrieved in multiple chunks, chunks are retrieved in sequence, where the length of each chunk is specified by the length parameter of each call to this routine. Chunks may only be retrieved sequentially through the character large object.

Parameters

```
int32_t MimerGetNclobData (
            MimerLob *clobhandle,
            wchar_t *data,
            size_t size)
```

`clobhandle`	in	A handle to a character large object returned by a call to `MimerGetLob`.
`data`	out	Where to place the entire or a part of the character large object.
`size`	in	The maximum number of bytes to retrieve, including terminating null. That is, if this parameter is `n`, the next `n-1` characters are retrieved from the large object.

Returns

A negative value indicating error, or a non-negative value indicating the number of characters available to return. This includes the number of characters returned in this call (excluding terminating null) and all data that are left in the character object. When the returned value is less than the length input parameter, all data of the character object has been returned.

This value may be used to calculate how many characters (excluding terminating zero) that were returned using this formula:

```
returned_characters = min(length_input_parameter-1,return_value)
```

The following formula may be used to calculate the number of characters left to return:

```
left_to_return = return_value-length_input_parameter+1
```

Return value	Description
> 0	The number of characters left to return.
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `clobhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerGetNclobData8

Retrieves the contents of a character large object. (UTF-8 version.)

A character large object may be retrieved in one or more chunks.

If the character large object is to be retrieved in multiple chunks, chunks are retrieved in sequence, where the length of each chunk is specified by the length parameter of each call to this routine. Chunks may only be retrieved sequentially through the character large object.

Parameters

```
int32_t MimerGetNclobData8 (
            size_t *returned,
            MimerLob clobhandle,
            char *data,
            size_t size)
```

`returned`	out	The number of bytes, excluding terminating zero, returned as a result of the operation.
`clobhandle`	in	A handle to a character large object returned by a call to MimerGetLob.
`data`	out	Where to place the entire or a part of the character large object.
`size`	in	The number of bytes to retrieve, including terminating null.

Returns

A negative value indicating error, or a non-negative value indicating the number of characters available to return. This includes the number of characters returned in this call (excluding terminating null) and all data that are left in the character object. When the returned value is less than the length input parameter, all data of the character object has been returned.

This value may be used to calculate how many characters (excluding terminating null) that were returned using this formula:

```
returned_characters = min(length_input_parameter-1,return_value)
```

The following formula may be used to calculate the number of characters left to return:

```
left_to_return = return_value-length_input_parameter+1
```

Return value	Description
>= 0	Success. The number of characters to return. (See description above.)
`MIMER_HANDLE_INVALID`	The `clobhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.

Return value	Description
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerGetNclobDataC

Retrieves the contents of a character large object. (char version.)

A character large object may be retrieved in one or more chunks.

If the character large object is to be retrieved in multiple chunks, chunks are retrieved in sequence, where the length of each chunk is specified by the length parameter of each call to this routine. Chunks may only be retrieved sequentially through the character large object.

Parameters

```
int32_t MimerGetNclobDataC (
            size_t *returned,
            MimerLob clobhandle,
            char *data,
            size_t size)
```

`returned`	out	The number of bytes, excluding terminating zero, returned as a result of the operation.
`clobhandle`	in	A handle to a character large object returned by a call to MimerGetLob.
`data`	out	Where to place the entire or a part of the character large object.
`size`	in	The number of bytes to retrieve, including terminating null.

Returns

A negative value indicating error, or a non-negative value indicating the number of characters available to return. This includes the number of characters returned in this call (excluding terminating null) and all data that are left in the character object. When the returned value is less than the length input parameter, all data of the character object has been returned.

This value may be used to calculate how many characters (excluding terminating null) that were returned using this formula:

```
returned_characters = min(length_input_parameter-1,return_value)
```

The following formula may be used to calculate the number of characters left to return:

```
left_to_return = return_value-length_input_parameter+1
```

Return value	Description
>= 0	Success. The number of characters to return. (See description above.)
`MIMER_HANDLE_INVALID`	The `clobhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.

Return value	Description
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerGetStatistics

Obtains server statistics information.

Statistics is returned in the form of counters. Counters may either be an absolute value representing the current status or a monotonically increasing value representing the number of occurred events since the server started. An example of the former is current number of users and an example of the latter is number of server page requests.

The available counter values are:

BSI_4K_PAGES	The number of 4 K pages available in the system.
BSI_32K_PAGES	The number of 32 K pages available in the system.
BSI_128K_PAGES	The number of 128 K pages available in the system.
BSI_PAGES_USED	The total number of pages in use.
BSI_4K_PAGES_USED	The number of 4 K pages in use.
BSI_32K_PAGES_USED	The number of 32 K pages in use.
BSI_128K_PAGES_USED	The number of 128 K pages in use.

Parameters

```
int32_t MimerGetStatistics (
            MimerSession sessionhandle,
            int32_t *counters,
            int16_t nr_of_counters)
```

sessionhandle	in	A session handle opened through a call to MimerBeginSession[C|8].
counters	inout	An array containing the counter values to retrieve. On output, the array contains the corresponding counter values. On return, a value of -2 indicates that the counter type was unknown, -1 indicate that the counter type was known but its value was not available.
nr_of_counters	in	Specifies the number of counters supplied in counters.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
MIMER_SUCCESS	Success.
MIMER_OUTOFMEMORY	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

MimerGetString

Gets character data from a result set or an output parameter. (wchar_t version.)

Only the SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER and NATIONAL CHARACTER VARYING may be retrieved using this function.

Parameters

```
int32_t MimerGetString (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            wchar_t *dest,
            size_t dest_maxlen)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`dest`	out	A memory location where to place output wide character data. The character data is null terminated. If this is a null pointer, no data will be returned, but the length is still returned by the routine.
`dest_maxlen`	in	The length of the `dest` memory location in characters.

Returns

A negative value indicating an error, or a zero or positive value indicating the number of characters in the column (not counting the terminating zero).

Return value	Description
>=0	Success. The number of characters to be returned. (If > `dest_maxlen`, output data was truncated.)
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER or NATIONAL CHARACTER VARYING data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.

Return value	Description
MIMER_SEQUENCE_ERROR	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
MIMER_SQL_NULL_VALUE	The column or output parameter contained the SQL null value. (This can also be detected using the MimerIsNull routine.)

Notes

Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerGetString8

Gets character data from a result set or an output parameter into a multi byte character string, where the character set is UTF-8.

Only the SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER and NATIONAL CHARACTER VARYING may be retrieved using this function.

Parameters

```
int32_t MimerGetString8 (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            char *dest,
            size_t dest_maxsiz)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`dest`	out	A memory location where to place UTF-8 character data. The character data is null terminated. If this is a null pointer, no data will be returned, but the length is still returned by the routine.
`dest_maxsiz`	in	The length of the `dest` memory location in bytes.

Returns

A negative value indicating an error, or a zero or positive value indicating the number of bytes in the column (not counting the terminating zero).

Return value	Description
>= 0	Success. The number of characters to be returned. (If > `dest_maxlen`, output data was truncated.)
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER or NATIONAL CHARACTER VARYING data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.

Return value	Description
MIMER_PARAMETER_NOT_OUTPUT	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
MIMER_SEQUENCE_ERROR	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
MIMER_SQL_NULL_VALUE	The column or output parameter contained the SQL null value. (This can also be detected using the MimerIsNull routine.)

Notes

Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerGetStringC

Gets character data from a result set or an output parameter into a multi byte character string, where the character set is defined by the current locale. (char version.)

Only the SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER and NATIONAL CHARACTER VARYING may be retrieved using this function.

Parameters

```
int32_t MimerGetStringC (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            char *dest,
            size_t dest_maxsiz)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to get data from. First parameter/column is 1.
`dest`	out	A memory location where to place locale dependent character data. The character data is null terminated. If this is a null pointer, no data will be returned, but the length is still returned by the routine.
`dest_maxsiz`	in	The length of the `dest` memory location in bytes.

Returns

A negative value indicating an error, or a zero or positive value indicating the number of bytes in the column (not counting the terminating zero).

Return value	Description
>= 0	Success. The number of characters to be returned. (If > `dest_maxlen`, output data was truncated.)
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER or NATIONAL CHARACTER VARYING data types.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.

Return value	Description
MIMER_PARAMETER_NOT_OUTPUT	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
MIMER_SEQUENCE_ERROR	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
MIMER_SQL_NULL_VALUE	The column or output parameter contained the SQL null value. (This can also be detected using the MimerIsNull routine.)

Notes

Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerGetUUID

Gets a Universally unique identifier from a result set or output parameter. Only the SQL data type BUILTIN.UUID can be retrieved using this function.

Parameters

```
int32_t MimerGetUUID (
            MimerStatement statementhandle,
            int16_t paramno_colno,
            unsigned char uuid[16])
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to check for null in. First column/parameter is 1.
`uuid`	out	A memory location where 16 bytes will be written corresponding to the identifier.

Returns

A negative value indicating an error or zero indicating success.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the output parameter or column was not of the BUILTIN.UUID data type.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.
`MIMER_SQL_NULL_VALUE`	The column or output parameter contained the SQL null value. (This can also be detected using the `MimerIsNull` routine.)

Notes

Micro API compatible.

MimerIsNull

Checks if a result set column or output parameter has the SQL null value.

Parameters

```
int32_t MimerIsNull (
            MimerStatement statementhandle,
            int16_t paramno_colno)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`paramno_colno`	in	The parameter number or column number to check for null in. First column/parameter is 1.

Returns

If the column or parameter is the SQL null value, a positive value is returned. If it is not the SQL null value zero is returned. If an error occurred, a negative error code is returned.

Return value	Description
> 0	Null is returned.
0	Not null.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The supplied column or parameter number did not exist.
`MIMER_PARAMETER_NOT_OUTPUT`	The referenced parameter is not an output or input/output parameter, which is required when calling a data output routine.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is not open, or the current row is before the first row of the result set, or after the last row of the result set.

Notes

Micro API compatible.

MimerNext

Advances the current row to the next row, but only within the current array.

If the end of the array is reached, end of data is returned, and a new request must be made to the server for additional rows.

The difference between calling this function and `MimerFetch` or `MimerFetchSkip` is that this routine is guaranteed to never make server requests and therefore guaranteed not to require any context switches.

Parameters

```
int32_t MimerNext (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement with an open cursor.

Returns

Returns `MIMER_SUCCESS` if the row was advanced one row. `MIMER_NO_DATA` was returned if the end of the array was reached.

Return value	Description
`MIMER_NO_DATA`	End of table reached.
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.

Notes

Not Micro API compatible.

MimerOpenCursor

Opens a cursor to be used when reading a result set.

Result sets are produced either by SELECT queries, or by calls to result set procedures.

All input parameters must be set prior to calling `MimerOpenCursor`, or else an error will occur.

If this routine returns successfully, the cursor is opened and positioned before the first row.

The first row can then be fetched by calling `MimerFetch`. After that column data may be retrieved using data output routines.

If the statement does not return a result set, this routine will return with a failure.

Parameters

```
int32_t MimerOpenCursor (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_SEQUENCE_ERROR`	If the statement cursor is already open.
`MIMER_UNSET_PARAMETER`	All parameters has not yet been set.

Notes

Micro API compatible.

MimerParameterCount

Obtains the number of parameters of a statement.

Parameters

```
int32_t MimerParameterCount (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying the compiled statement which parameters to count.

Returns

If zero or positive, the number of parameters. If negative a standard Mimer error code.

Return value	Description
> = 0	Number of parameters.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	The statement is not compiled.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

MimerParameterMode

Detects the input/output mode of a parameter.

Parameters

```
int32_t MimerParameterMode (
            MimerStatement statementhandle,
            int16_t paramno)
```

`statementhandle`	in	The statement whose parameter to look up.
`paramno`	in	The parameter to look up, where the first one is number 1.

Returns

If zero or positive, the number of parameters. If negative a standard Mimer error code.

Return value	Description
1	The parameter was input.
2	The parameter was output.
3	The parameter was input/output.
`MIMER_SEQUENCE_ERROR`	The statement was not in a prepared state.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The parameter does not exist.

Notes

Not Micro API compatible.

MimerParameterName

Obtains the name of a parameter. (wchar_t version.)

Parameters

```
int32_t MimerParameterName (
            MimerStatement statementhandle,
            int16_t parameter,
            wchar_t *parametername,
            size_t maxlen)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying the compiled statement which parameters to count.
`parameter`	in	The parameter number, where the leftmost parameter is 1.
`parametername`	out	The area where to return the parameter name. The maximum parameter name length returned is maxlen-1.
`maxlen`	in	The length of the parametername area in characters, including room for terminating null.

Returns

Returns the number of characters in the parameter name. If this value is larger than maxlen-1, a truncation occurred. If a negative value was returned, there was an error.

Return value	Description
> = 0	Number of characters in parameter name.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	The statement is not compiled.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The parameter number does not exist.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerParameterName8

Obtains the name of a parameter. (UTF-8 version.)

Parameters

```
int32_t MimerParameterName8 (
            MimerStatement statementhandle,
            int16_t parameter,
            char *parametername,
            size_t maxsize)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying the compiled statement which parameters to count.
`parameter`	in	The parameter number, where the leftmost parameter is 1.
`parametername`	out	The area where to return the parameter name. The maximum parameter name length returned is maxsize-1.
`maxsize`	in	The length of the parametername area in bytes, including room for terminating null.

Returns

Returns the number of bytes in the parameter name. If this value is larger than maxsize-1, a truncation occurred. If a negative value was returned, there was an error.

Return value	Description
> = 0	Number of bytes in parameter name.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	The statement is not compiled.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The parameter number does not exist.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerParameterNameC

Obtains the name of a parameter. (char version.)

Parameters

```
int32_t MimerParameterNameC (
            MimerStatement statementhandle,
            int16_t parameter,
            char *parametername,
            size_t maxsize)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying the compiled statement which parameters to count.
`parameter`	in	The parameter number, where the leftmost parameter is 1.
`parametername`	out	The area where to return the parameter name. The maximum parameter name length returned is maxsize-1.
`maxsize`	in	The length of the parametername area in bytes, including room for terminating null.

Returns

Returns the number of bytes in the parameter name. If this value is larger than maxsize-1, a truncation occurred. If a negative value was returned, there was an error.

Return value	Description
> = 0	Number of bytes in parameter name.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	The statement is not compiled.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The parameter number does not exist.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

Not Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerParameterType

Obtains the data type of a parameter.

Parameters

```
int32_t MimerParameterType(
            MimerStatement statementhandle,
            int16_t parameter)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying the compiled statement which parameters to count.
`parameter`	in	The parameter number, where the leftmost parameter is 1.

Returns

Returns the parameter type or a negative value if an error occurred.

Return value	Description
> 0	Data type. See list in *Data Types* on page *103*
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	Statement is not compiled.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The parameter number does not exist.

Notes

Not Micro API compatible.

MimerRowSize

Returns the maximum number of bytes required to hold one row of data. This routine might be used to calculate the maximum number of rows allowed in an array fetching scenario under certain memory restrictions.

For example, if it was determined that the fetching buffer must use no more than 20 000 bytes, although it would be desirable to use array fetching for performance reasons, calling this function would obtain a value to divide 20 000 with obtaining a reasonable maximum array set size.

Parameters

```
int32_t MimerRowSize (MimerStatement statementhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying the compiled statement which maximum row size to obtain.

Returns

If positive, the minimum number of bytes required to be guaranteed to hold one row of data. If negative, an error condition.

Return value	Description
>= 0	Number of bytes required to hold one row of data. 0 means that no input data
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	Statement is not compiled.

Notes

Not Micro API compatible.

MimerSetArraySize

Set the array size.

By default the Mimer API routines `MimerFetch` and `MimerFetchSkip` uses an internal fetch buffer equal to the maximum size of one row. Depending on the actual size of the data, this buffer may hold more than one row. By increasing the array size, more data is retrieved in each server request.

Parameters

```
int32_t MimerSetArraySize (
            MimerStatement statementhandle,
            int32_t arraysize)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying the compiled statement which array size to change.
`arraysize`	in	The number of rows to retrieve in each request.

Returns

`MIMER_SUCCESS` if successful. A negative value indicate an error.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_SEQUENCE_ERROR`	Statement is not compiled.

Notes

Not Micro API compatible.

MimerSetBinary

Sets a binary data parameter.

Only the SQL types BINARY and BINARY VARYING may be set using this call.

This routine may allocate additional memory to hold the parameter value, along with future output parameters and result set columns. This memory is freed when it is no longer needed, no later than when `MimerEndStatement` is called. This memory is also freed after a call to `MimerExecute` and `MimerCloseCursor` if there were no output parameters or result set columns.

Parameters

```
int32_t MimerSetBinary (
            MimerStatement statementhandle,
            int16_t param_no,
            const void *value,
            size_t size)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A pointer to a memory location holding the binary data. (If this pointer is a null pointer, the parameter is set to the SQL null value.)
`size`	in	The maximum size of the binary data in bytes. To set the parameter to the SQL null value, use the `MimerSetNull` function.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not BINARY or BINARY VARYING.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.

Return value	Description
MIMER_PARAMETER_NOT_INPUT	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.
MIMER_TRUNCATION_ERROR	The input data was truncated.

Notes

Not Micro API compatible.

MimerSetBlobData

Sets the data of a binary large object.

Whenever a binary large object has been created using a call to `MimerSetLob`, it has to be filled with data. This is done using a call to this routine. A call to this routine must always be preceded by a call to `MimerSetLob`.

The total length of all the chunks supplied by one or more calls to this routine must coincide with the length supplied in the preceding call to `MimerSetLob`, or else an error will occur.

When the entire binary large object has been set, the handle is automatically released.

Parameters

```
int32_t MimerSetBlobData (
            MimerLob *blobhandle,
            const void *data,
            size_t size)
```

`blobhandle`	in	A reference to a binary large object handle created by `MimerSetLob`.
`data`	in	A pointer to the binary large object data.
`size`	in	Size of the chunk supplied in this call, in bytes. The size of each chunk is limited to slightly less than 10 MB. If larger objects than this is to be set, multiple calls to this routine must be used, each supplying the large object in chunks.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `blobhandle` parameter was not recognized as a handle.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

MimerSetBoolean

Sets a boolean data parameter.

Only the database type BOOLEAN may be set using this call.

Parameters

```
int32_t MimerSetBoolean (
            MimerStatement statementhandle,
            int16_t param_no,
            int32_t value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A BOOLEAN value. `0` = false `1` = true

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not BOOLEAN.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Notes

Micro API compatible.

MimerSetDouble

Sets a double precision floating point parameter.

Only the SQL data types REAL and DOUBLE PRECISION may be set using this call.

This routine may allocate additional memory to hold the parameter value, along with future output parameters and result set columns. This memory is freed when it is no longer needed, no later than when `MimerEndStatement` is called. This memory is also freed after a call to `MimerExecute` and `MimerCloseCursor` if there were no output parameters or result set columns.

Parameters

```
int32_t MimerSetDouble (
            MimerStatement statementhandle,
            int16_t param_no,
            double value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A double precision floating point value. To set the parameter to the SQL null value, use the `MimerSetNull` function.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not FLOAT, REAL or DOUBLE PRECISION.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Return value	Description
MIMER_UNDEFINED_FLOAT_VALUE	The supplied double was either a not-a-number or infinity.

Notes

Micro API compatible.

MimerSetFloat

Sets a single precision floating point parameter.

Only the SQL data types REAL and DOUBLE PRECISION may be set using this call.

This routine may allocate additional memory to hold the parameter value, along with future output parameters and result set columns. This memory is freed when it is no longer needed, no later than when `MimerEndStatement` is called. This memory is also freed after a call to `MimerExecute` and `MimerCloseCursor` if there were no output parameters or result set columns.

Parameters

```
int32_t MimerSetFloat (
            MimerStatement statementhandle,
            int16_t param_no,
            float value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A single precision floating point value. To set the parameter to the SQL null value, use the `MimerSetNull` function.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not REAL or DOUBLE PRECISION.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.
`MIMER_UNDEFINED_FLOAT_VALUE`	The supplied double was either a not-a-number or infinity.

Notes

Micro API compatible.

MimerSetInt32

Sets an integer parameter.

Only the SQL data types INTEGER, BIGINT and SMALLINT may be set using this call.

This routine may allocate additional memory to hold the parameter value, along with future output parameters and result set columns. This memory is freed when it is no longer needed, no later than when `MimerEndStatement` is called. This memory is also freed after a call to `MimerExecute` and `MimerCloseCursor` if there were no output parameters or result set columns.

Parameters

```
int32_t MimerSetInt32 (
            MimerStatement statementhandle,
            int16_t param_no,
            int32_t value)
```

statementhandle	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
param_no	in	A number identifying the parameter. First parameter is 1.
value	in	An integer value. To set the parameter to the SQL null value, use the `MimerSetNull` function.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not an INTEGER, BIGINT or SMALLINT parameter.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Notes

Micro API compatible.

MimerSetInt64

Sets an int64_t (long long) parameter.

Only the SQL data types INTEGER, BIGINT and SMALLINT may be set using this call.

This routine may allocate additional memory to hold the parameter value, along with future output parameters and result set columns. This memory is freed when it is no longer needed, no later than when `MimerEndStatement` is called. This memory is also freed after a call to `MimerExecute` and `MimerCloseCursor` if there were no output parameters or result set columns.

Parameters

```
int32_t MimerSetInt64 (
            MimerStatement statementhandle,
            int16_t param_no,
            int64_t value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	An integer value. To set the parameter to the SQL null value, use the `MimerSetNull` function.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not an integer parameter.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Notes

Micro API compatible.

MimerSetLob

Sets a large object in the database.

This routine is used to create a large object (BINARY LARGE OBJECT, a CHARACTER LARGE OBJECT, or a NATIONAL CHARACTER LARGE OBJECT) in the database.

Whenever a large object is about to the inserted in the database as a parameter to a statement, this routine is called for that specific parameter.

Upon successful return, a handle to a large object is stored in the database. The contents of the object may be set at a later time using one or more calls to `MimerSetBlobData` or `MimerSetNclobData`.

If the size/length of the object is 0, `MimerSetBlobData` or `MimerSetNclobData` need not be called. If the size is larger than 0, `MimerSetBlobData` or `MimerSetNclobData` must be called to complete the creation of the large object.

To set the parameter to the SQL null value, use the `MimerSetNull` function.

Parameters

```
int32_t MimerSetLob (
            MimerStatement statementhandle,
            int16_t param_no,
            size_t size_length,
            MimerLob *lobhandle)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`size_length`	in	The size of the BINARY LARGE OBJECT in bytes, or the length of CHARACTER LARGE OBJECT or NATIONAL CHARACTER LARGE OBJECT in characters.
`lobhandle`	out	The handle to the created large object. If the object was of size 0, a handle is not created. This condition may be detected by comparing with the symbol `MIMERNULLHANDLE`.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not a BINARY LARGE OBJECT, a CHARACTER LARGE OBJECT or a NATIONAL CHARACTER LARGE OBJECT.

Return value	Description
MIMER_HANDLE_INVALID	The statementhandle parameter was not recognized as a handle.
MIMER_NONEXISTENT_COLUMN_PARAMETER	The referenced parameter does not exist.
MIMER_NULL_VIOLATION	Cannot assign the null value to a non-nullable parameter
MIMER_OUTOFMEMORY	If not enough memory could be allocated.
MIMER_PARAMETER_NOT_INPUT	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Notes

Micro API compatible.

MimerSetNclobData

Sets the data of a character large object. (wchar_t version.)

Whenever a character large object has been created using a call to `MimerSetLob`, it has to be filled with data. This is done using a call to this routine. A call to this routine must always be preceded by a call to `MimerSetLob`.

When the entire character large object has been set, the handle is automatically released.

Parameters

```
int32_t MimerSetNclobData (
            MimerLob *clobhandle,
            const wchar_t *data,
            size_t length)
```

`clobhandle`	in	A reference to a character large object handle created by `MimerSetLob`.
`data`	in	A pointer to the character data.
`length`	in	Length of the chunk supplied in this call, in characters. The size of each chunk is limited to slightly less than 10 MB. This translates to, in the worst case, about 2.5 million characters. If larger character objects than this is to be set, multiple calls to this routine must be used, each supplying the character large object in chunks.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `clobhandle` parameter was not recognized as a handle.
`MIMER_ILLEGAL_CHARACTER`	The input string contained illegal characters.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetNclobData8

Sets the data of a character large object. (UTF-8 version.)

Whenever a character large object has been created using a call to `MimerSetLob`, it has to be filled with data. This is done using a call to this routine. A call to this routine must always be preceded by a call to `MimerSetLob`.

When the entire character large object has been set, the handle is automatically released.

Parameters

```
int32_t MimerSetNclobData8 (
            MimerLob *clobhandle,
            const char *data,
            size_t size)
```

`clobhandle`	in	A reference to a character large object handle created by `MimerSetLob`.
`data`	in	A pointer to the character data.
`size`	in	Length of the chunk supplied in this call, in bytes. The size of each chunk is limited to slightly less than 10 MB. This translates to, in the worst case, about 2.5 million characters. If larger character objects than this is to be set, multiple calls to this routine must be used, each supplying the character large object in chunks.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Not Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetNclobDataC

Sets the data of a character large object using a multibyte character pointer. (char version.)

Whenever a character large object has been created using a call to `MimerSetLob`, it has to be filled with data. This is done using a call to this routine. A call to this routine must always be preceded by a call to `MimerSetLob`.

When the entire character large object has been set, the handle is automatically released.

Parameters

```
int32_t MimerSetNclobDataC (
            MimerLob *clobhandle,
            const char *data,
            size_t size)
```

`clobhandle`	in	A reference to a character large object handle created by `MimerSetLob`.
`data`	in	A pointer to the character data.
`size`	in	Length of the chunk supplied in this call, in bytes. The size of each chunk is limited to slightly less than 10 MB. This translates to, in the worst case, about 2.5 million characters. If larger character objects than this is to be set, multiple calls to this routine must be used, each supplying the character large object in chunks.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
Other value < 0	Any of the server error codes listed in *Appendix B Return Codes*.

Notes

This routine interacts with the database server.

Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetNull

Sets an input parameter to the SQL null value.

Parameters

```
int32_t MimerSetNull (
            MimerStatement statementhandle,
            int16_t paramno)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`paramno`	in	The number of the parameter to set to null. First column/parameter is 1.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Notes

Micro API compatible.

MimerSetString

Sets a string parameter. (wchar_t version.)

This call sets a string parameter of a statement call. The SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL and NUMERIC may be set using this call.

Parameters

```
int32_t MimerSetString (
            MimerStatement statementhandle,
            int16_t param_no,
            const wchar_t *value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A pointer to a buffer holding a null terminated character string. If this pointer is a null pointer, the parameter is set to the SQL null value.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred SQL data type was not a CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL or a NUMERIC.
`MIMER_ILLEGAL_CHARACTER`	If the input string contained illegal Unicode characters.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Return value	Description
MIMER_TRUNCATION_ERROR	The input string was longer than could be held in the parameter.

Notes

Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetString8

Sets a string parameter using a UTF-8 character string.

The SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL and NUMERIC may be set using this call.

Parameters

```
int32_t MimerSetString8 (
            MimerStatement statementhandle,
            int16_t param_no,
            const char *value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A pointer to a buffer holding a null terminated character string. If this pointer is a null pointer, the parameter is set to the SQL null value.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred SQL data type was not a CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL or a NUMERIC.
`MIMER_ILLEGAL_CHARACTER`	If the input string contained illegal characters.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.

Return value	Description
MIMER_PARAMETER_NOT_INPUT	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.
MIMER_TRUNCATION_ERROR	The input string was longer than could be held in the parameter.

Notes

Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetStringC

Sets a string parameter using a multibyte character string.

The SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL and NUMERIC may be set using this call.

Parameters

```
int32_t MimerSetStringC (
            MimerStatement statementhandle,
            int16_t param_no,
            const char *value)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A pointer to a buffer holding a null terminated character string. If this pointer is a null pointer, the parameter is set to the SQL null value.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred SQL data type was not a CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL or a NUMERIC.
`MIMER_ILLEGAL_CHARACTER`	If the input string contained illegal characters.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.

Return value	Description
MIMER_PARAMETER_NOT_INPUT	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.
MIMER_TRUNCATION_ERROR	The input string was longer than could be held in the parameter.

Notes

Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetStringLen

Sets a string parameter. (wchar_t version.)

This call sets a string parameter of a statement call. The SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL and NUMERIC may be set using this call.

Parameters

```
int32_t MimerSetStringLen (
            MimerStatement statementhandle,
            int16_t param_no,
            const wchar_t *value,
            size_t length)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A pointer to a buffer holding a null terminated wide character string. If this pointer is a null pointer, the parameter is set to the SQL null value.
`length`	in	Length of `value` in characters.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred SQL data type was not a CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL or a NUMERIC.
`MIMER_ILLEGAL_CHARACTER`	If the input string contained illegal characters.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.

Return value	Description
MIMER_PARAMETER_NOT_INPUT	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.
MIMER_TRUNCATION_ERROR	The input string was longer than could be held in the parameter.

Notes

Not Micro API compatible.

wchar_t version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetStringLen8

Sets a string parameter using a UTF-8 character string. (UTF-8 version.)

The SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL and NUMERIC may be set using this call.

Parameters

```
int32_t MimerSetStringLen8 (
            MimerStatement statementhandle,
            int16_t param_no,
            const char *value,
            size_t length)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A pointer to a buffer holding a null terminated character string. If this pointer is a null pointer, the parameter is set to the SQL null value.
`length`	in	Size of `value` in bytes.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred SQL data type was not a CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL or a NUMERIC.
`MIMER_ILLEGAL_CHARACTER`	If the input string contained illegal Unicode characters.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Return value	Description
MIMER_TRUNCATION_ERROR	The input string was longer than could be held in the parameter.

Notes

Not Micro API compatible.

UTF-8 version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetStringLenC

Sets a string parameter using a multibyte character pointer. (char version.)

The SQL data types CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL and NUMERIC may be set using this call.

Parameters

```
int32_t MimerSetStringLenC (
            MimerStatement statementhandle,
            int16_t param_no,
            const char *value,
            size_t length)
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]` identifying a prepared statement.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`value`	in	A pointer to a buffer holding a null terminated character string. If this pointer is a null pointer, the parameter is set to the SQL null value.
`length`	in	Size of `value` in bytes.

Returns

A negative value indicating an error, or zero if successful.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred SQL data type was not a CHARACTER, CHARACTER VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, DATE, TIME, TIMESTAMP, DECIMAL or a NUMERIC.
`MIMER_ILLEGAL_CHARACTER`	If the input string contained illegal Unicode characters.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Return value	Description
MIMER_TRUNCATION_ERROR	The input string was longer than could be held in the parameter.

Notes

Not Micro API compatible.

char version of the routine. See *Character String Formats* on page *98* for more information about character formats and the different routine versions.

MimerSetUUID

Sets a Universally unique identifier parameter. Only the SQL data type BUILTIN.UUID can be set using this function.

Parameters

```
int32_t MimerSetUUID (
            MimerStatement statementhandle,
            int16_t param_no,
            const unsigned char uuid[16])
```

`statementhandle`	in	A handle returned by `MimerBeginStatement[C|8]`, identifying a statement that have been executed.
`param_no`	in	A number identifying the parameter. First parameter is 1.
`uuid`	in	A pointer to a 16 byte buffer where the identifier to set is located. If this pointer is NULL, the SQL NULL value may be set.

Returns

A negative value indicating an error or zero indicating success.

Return value	Description
`MIMER_SUCCESS`	Success.
`MIMER_CAST_VIOLATION`	If the referred database type was not of the BUILTIN.UUID type.
`MIMER_HANDLE_INVALID`	The `statementhandle` parameter was not recognized as a handle.
`MIMER_NONEXISTENT_COLUMN_PARAMETER`	The referenced parameter does not exist.
`MIMER_NULL_VIOLATION`	Cannot assign the null value to a non-nullable parameter.
`MIMER_OUTOFMEMORY`	If not enough memory could be allocated.
`MIMER_PARAMETER_NOT_INPUT`	The referenced parameter is not an input or input/output parameter, which is required when calling a data input routine.

Notes

Micro API compatible.

Chapter 8

Idents and Privileges

This chapter discusses Mimer SQL idents, privileges and database connections.

Mimer SQL Idents

An ident is an authorization-ID used to identify different kinds of users in a Mimer SQL database.

There are three kinds of ident in Mimer SQL: `USER`, `PROGRAM` and `GROUP`.

Idents connect to a database through the `CONNECT` statement and the `ENTER` statement is used to take up the privileges provided by a `PROGRAM` ident (see below).

Every `USER` ident has a unique ident name and an optional private password which must be correctly supplied to the `CONNECT` or `ENTER` statement in application programs. A `USER` ident may access the database without explicitly providing a username or password on condition that the username for the user currently logged in to the operating system correspond to the definition of the `USER` in the Mimer SQL database.

Every `PROGRAM` ident has a unique ident name and a private password which must be correctly supplied to the `CONNECT` or `ENTER` statement in application programs.

USER

`USER` idents are authorized to connect to a Mimer SQL database by using the `CONNECT` statement in an application program, or by entering the correct ident name and password in an interactive environment, or by using an `OS_USER` login.

Any privileges a `USER` ident holds may be exercised once the ident has logged on. `USER` idents are generally associated with specific physical individuals authorized to connect to the database.

PROGRAM

`PROGRAM` idents provide specific privileges required when executing certain operations. `PROGRAM` idents may not initiate a connection to a Mimer SQL database, but may be entered from within an application program or interactive environment by using the `ENTER` statement.

A connection should have been established before the ENTER statement is used, see *Chapter 4, Connecting to a Database*. Entering a PROGRAM ident is analogous to logging on as a USER ident, in that the PROGRAM ident gains access to the system and any privileges the ident holds become applicable.

PROGRAM idents are generally associated with specific functions within the system, not with physical individuals.

GROUP

GROUP idents are collective identities that provide common privileges for groups of users or PROGRAM idents.

Any privileges granted to or revoked from a GROUP ident automatically apply to all members of the group.

Any ident can be a member of as many groups as required, and one group can include any number of members.

GROUP idents provide a facility for organizing the privilege structure in the database system.

Database Privileges

The access of each ident to the database is defined by privileges granted within the system.

The privileges are grouped as follows:

- System privileges
- Object privileges
- Access privileges.

System Privileges

Privilege	Explanation
BACKUP	gives the right to perform databank backup and restore operations.
DATABANK	gives the right to create databanks.
IDENT	gives the right to create idents.
SCHEMA	gives the right to create schemas.
SHADOW	gives the right to create and manage databank shadows.
STATISTICS	gives the right to execute the UPDATE STATISTICS statement.

Object Privileges

Privilege	Explanation
TABLE	gives the right to create tables in a specified databank.

Privilege	Explanation
SEQUENCE	gives the right to create sequences in a specified databank.
EXECUTE	gives the right to access a routine or to enter (connect to) a specified PROGRAM ident.
MEMBER	grants membership in a specified GROUP ident.
USAGE	gives the right to specify the named domain where a data type would normally be specified (in contexts where use of domains is allowed) or the right to use a specified sequence or collation.

Access Privileges

Privilege	Explanation
SELECT	gives the right to read the table contents.
INSERT	gives the right to add new rows to the table.
DELETE	gives the right to remove rows from the table.
UPDATE	gives the right to update the contents of the table.
REFERENCES	gives the right to use the primary key or unique keys of the table as a foreign key from another table.

About Privileges

System privileges are automatically granted to the system administrator at installation, and may be passed on to other idents.

Object and access privileges are initially granted only to the creator of an object. The creator may however grant the privileges on to other idents.

All privileges may be granted with or without GRANT OPTION, which controls the right of the receiving ident to grant the privilege on to another ident.

Certain operations are not controlled by explicit privileges, but may only be performed by the creator of the object involved. These operations include ALTER (with the exception of ALTER IDENT, which may be performed by either the ident himself or by the creator of the ident), DROP and COMMENT. Similarly, privileges may only be revoked by their grantor.

Chapter 9

Transaction Handling and Database Security

This chapter explains transaction principles, transaction control statements, logging and protecting against data loss.

Transaction Principles

A transaction is an atomic operation which may not be divided into smaller operations.

Three transaction phases exist: build-up, during which the database operations are requested; prepare, during which the transaction is validated; commitment, during which the operations performed in the transaction are written to disk.

Read-only transactions have only two phases: build-up and prepare.

Transaction build-up may be started explicitly or implicitly, see *Starting Transactions* on page *225*; prepare and commitment are both initiated explicitly through a request to commit the transaction (using `COMMIT`).

In interactive application programs, build-up takes place typically over a time period determined by the user, while prepare and commitment are part of the internal process of committing a transaction, which occurs on a time-scale determined by machine operations.

Optimistic Concurrency Control

Since Mimer SQL uses optimistic concurrency control (OCC), deadlocks never occur, see *Locking* on page *223* for a further discussion of deadlocks. How optimistic concurrency control works in Mimer SQL is described below.

The transaction begins by taking a snapshot of the database in a consistent state. During build-up, changes requested to the contents of the database are kept in a write-set and are not visible to other users of the system. This allows the database to remain fully accessible to all users. The application program in which build-up occurs will see the database as though the changes had already been applied. Changes requested during transaction build-up become visible to other users when the transaction is successfully committed.

During build-up, a read-set records the state of the database as seen at the time of each operation (including intended changes). If the state of the database at commitment is inconsistent with the read-set, a conflict is reported and the transaction is rolled back (i.e. the write-set is erased and no changes are made to the database). This can happen if, for instance, a transaction updates a row which gets deleted by another user after build-up has started but before the transaction is committed. The application program is responsible for taking appropriate action if a transaction conflict occurs.

Concurrency Control Guidelines

Because of the nature of this concurrency control protocol, it is important that some of the implications are understood.

A transaction that exists for a long elapsed time has a greater chance of conflicting with changes made by other users than a transaction with a short elapsed time.

At the other extreme, an application that immediately commits every executed SQL statement will seldom meet any conflicts, but will incur unnecessary overhead.

In general:

- keep transactions as short as is reasonably possible
- keep interactive user dialogs outside of transactions

A common situation that can generate unnecessarily large read-sets is the following: an application program reads through the rows in a table in a loop construct, with a conditional exit to update a row on user intervention.

It is tempting to simply place a `COMMIT` after the update statement, for example:

```
EXEC SQL DECLARE c_1 CURSOR FOR SELECT…
loop
    EXEC SQL FETCH c_1
        INTO :VAR1, :VAR2, ..., :VARn;
    display VAR1, VAR2, ..., VARn;
    prompt "Update row?";
    exit when ANSWER = "YES";
end loop;
EXEC SQL UPDATE table
         SET ...
         WHERE CURRENT OF c_1;
EXEC SQL COMMIT;
```

However, the `FETCH` loop can create a large read-set while waiting for the user update request, risking transaction conflict at the `UPDATE`.

A tempting solution for this problem might be:

```
EXEC SQL DECLARE c_1 CURSOR FOR SELECT…
loop
    EXEC SQL FETCH c_1
        INTO :VAR1, :VAR2, ..., :VARn;
    display VAR1, VAR2, ..., VARn;
    prompt "Update row?";
    exit when ANSWER = "YES";
    EXEC SQL ROLLBACK;
end loop;
EXEC SQL UPDATE table
         SET ...
         WHERE CURRENT OF c_1;
EXEC SQL COMMIT;
```

But since `ROLLBACK` closes all cursors, this will not work.

Instead, something like the following is a better approach:

```
EXEC SQL DECLARE c_1 CURSOR FOR SELECT…
EXEC SQL SET TRANSACTION READ ONLY;
loop
    EXEC SQL FETCH c_1
        INTO :VAR1, :VAR2, ..., :VARn;
    display VAR1, VAR2, ..., VARn;
    prompt "Update row?";
    exit when ANSWER = "YES";
end loop;
EXEC SQL ROLLBACK;
EXEC SQL UPDATE table
         SET ...
         WHERE col1 = :VAR1,
               col2 = :VAR2, ...
EXEC SQL COMMIT;
```

The risk of a transaction conflict in the final transaction is minimal, because both the size and duration of the transaction is minimized. The use of a read-only transaction can significantly improve the performance of the `FETCH` statements.

A number of changes are necessary if we want to loop over `FETCH`, `UPDATE` and `COMMIT`.

```
EXEC SQL DECLARE c_1 CURSOR WITH HOLD FOR SELECT…
loop
    EXEC SQL FETCH c_1
        INTO :VAR1, :VAR2, ..., :VARn;
    display VAR1, VAR2, ..., VARn;
    prompt "Update row?";
    if  ANSWER = "YES" then
        EXEC SQL COMMIT;
        EXEC SQL UPDATE table
                 SET ...
                 WHERE col1 = :VAR1,
                       col2 = :VAR2, ...
        EXEC SQL COMMIT;
    end if;
end loop;
```

The cursor is declared `WITH HOLD` in order to remain open and positioned after `COMMIT`.

`COMMIT` is used instead of `ROLLBACK`, since holdable cursors does not remain open after `ROLLBACK`.

The `SET TRANSACTION` statement is removed, because the existence of an open holdable cursor prohibits a change of transaction mode. The cursor cannot be accessed both in `READ ONLY` and in `READ WRITE` mode.

Locking

Deadlock situations, which can be relatively common in some database management systems where records are locked during transaction build-up, can not occur in Mimer SQL.

In Mimer SQL it is impossible for two processes to be waiting for a record locked by the other process. In some other database management systems this situation may require operator intervention to resolve the problem.

In any database system, at some stage in a transaction, the data records must be locked to prevent access by other processes and to ensure that the transaction is not interrupted. In the Mimer SQL system, no change is made to the database contents during the transaction build-up and no records are locked. This means that the database can be freely accessed (and updated) by any other process; the data accessed by the transaction is only locked during the commit phase. In this way, locks are held only for a very short period of time.

The problems associated with locking are further reduced since only those records that are actually to be updated are locked. Other data in the same table continues to be accessible to other transactions.

Transactions and Logging

Changes made to a database may be logged, to provide back-up protection in the event of hardware failure, provided that the changes occur within a transaction and that the databanks involved have the `LOG` option. Transaction handling is, therefore, important even in standalone environments where concurrency control issues do not arise.

Options

Transaction control and logging is determined at the databank level by options set when the databank is defined. The options are:

- **LOG**

  All operations on the databank are performed under transaction control. All transactions are logged.

- **TRANSACTION**

  All operations on the databank are performed under transaction control. No transactions are logged.

- **WORK**

  All operations on the databank are performed without transaction control (even if they are requested within a transaction), and are not logged. Sets of operations (`DELETE`, `UPDATE` or `INSERT` on several rows) that are interrupted will not be rolled back.

- **READ ONLY**

  Only read operations are allowed, i.e no `UPDATE`, `DELETE` or `INSERT` can be performed.

Note: All important databanks should be defined with `LOG` option, so that valuable data is not lost by any system failure.

Protecting Against Data Loss

The following sections discuss how system interruptions and hardware failure are handled.

System Interruptions

If a transaction build-up is interrupted by a system failure or a program termination (deliberate or otherwise) the transaction is aborted and none of the requested changes are made to the database.

Transactions which are interrupted after the request to commit, but before all operations in the transaction have been executed on the database, are completed by the automatic recovery functionality when the databank involved is next accessed. There is no possibility of transaction conflict in such an automatic completion, since no other process can access the affected data as long as an incomplete transaction is pending.

In the event of a system failure that interrupts one or more application programs, it may be necessary to manually examine the database contents to determine which transactions failed to commit before the interruption.

Hardware Failure

A databank that is damaged by hardware failure (e.g. a disk crash) may be recovered using back-up copies and the transaction log (`LOGDB`), provided that all write operations on the databank have been logged (and that the back-up copies and the `LOGDB` databank are intact).

Backup and restore facilities are described in the *Mimer SQL System Management Handbook*.

Transaction Control Statements

The following sections explain how to start, end and optimize transactions. You can also read about consistency and exception diagnostics in transactions. Further, transaction options, cursors and error handling are discussed.

Starting Transactions

Transaction start may be set to `EXPLICIT` or `IMPLICIT`.

The default transaction start setting is `IMPLICIT`, which means a transaction will be started automatically whenever one is needed.

To set the transaction start mode, use the statements:

```
SET TRANSACTION START EXPLICIT;

SET TRANSACTION START IMPLICIT;
```

Different database connections can use different transaction start options.

The START statement can always be used to explicitly start a transaction. This is useful if a number of related updates are to be performed and it is desirable that all the updates succeed or fail together to maintain consistency.

You cannot start a transaction while a transaction is already active.

Explicit Transaction Start

With this setting, transactions are never automatically started. All transactions must be explicitly started by executing the START statement.

Any update operation (INSERT, UPDATE or DELETE) involving a table in a databank with the TRANS or LOG option must occur within a transaction. An error will be raised if such an update is attempted without first starting a transaction.

All the statements issued after the START statement and before the transaction is concluded are grouped together within that single transaction.

A transaction is concluded by executing a COMMIT or ROLLBACK statement.

Implicit Transaction Start

With this setting, a transaction is started automatically (if one is not already active) by a reference to an object stored in a databank with the TRANS or LOG option (i.e. if none of the objects referenced are stored in a databank with the TRANS or LOG option, no transaction is required and therefore one is not started).

The START statement may be used to explicitly start a transaction if required, typically to allow several updates to be grouped together within a single transaction for consistency, as already described.

An automatically started transaction is concluded by executing a COMMIT or ROLLBACK statement.

All the statements issued after the initiating update and before the concluding COMMIT or ROLLBACK statement are grouped together within that single transaction.

Ending Transactions

Transactions must be ended with the COMMIT or ROLLBACK statement.

- **COMMIT**

  This statement requests that the operations in the write-set are executed on the database, making the changes permanent and visible to other users. The SQLSTATE value returned when a COMMIT statement is executed indicates either that the transaction commitment was successful (SQLSTATE = '00000') or that a transaction conflict occurred (SQLSTATE <> '00000').

- **ROLLBACK**

  This statement abandons the transaction. The read-set and write-set are dropped and no changes are made to the database. ROLLBACK is always successful.

  Note: A transaction in Mimer SQL is never physically rolled-back in the sense of undoing changes made to the database, since changes are not actually effected until a successful COMMIT is performed. However, the ROLLBACK statement may free internal resources.

Transactions that are not successfully committed due to a transaction conflict do not have to be explicitly rolled back. The `ROLLBACK` statement is most commonly used in exception routines for handling error situations that are detected by the application during transaction build-up.

If a connection or program is terminated without requesting a `COMMIT` or `ROLLBACK` for the current transaction, the system will abort the transaction. None of the changes requested during the transaction build-up will be made to the database.

Transaction handling in BSQL differs slightly from that described here – see the *Mimer SQL User's Manual, Chapter 6, Handling Transactions*, for details.

Optimizing Transactions

The following `SET TRANSACTION` options are used to optimize transaction performance:

- **READ ONLY**

  This setting should always be used for transactions that do not require update access to the database. Significant performance gains can be achieved, especially for queries retrieving large numbers of rows, when this setting is used in queries when there is no need for update access to the database.

- **READ WRITE**

  This setting should only be used for transactions that require update access to the database. This is the default setting for a transaction.

The default option is `READ WRITE`, or the option defined to be the default for the current session by using the `SET SESSION` statement, see *Setting Default Transaction Options* on page *228*.

The `SET TRANSACTION READ` command only affects the single next transaction started after it is used.

Consistency Within Transactions

The `SET TRANSACTION … ISOLATION LEVEL` options can be used to control the degree to which the changes occurring within one transaction are affected by the changes occurring within other concurrently executing transactions.

The default option is `REPEATABLE READ`, or the option defined to be the default for the current session by using the `SET SESSION` statement, see *Setting Default Transaction Options* on page *228*.

The `SET TRANSACTION … ISOLATION LEVEL` command only affects the single next transaction started after it is used.

Options

The following options are available:

- **SERIALIZABLE**

  This setting guarantees that the end result of the operations performed by two or more concurrent transactions will be the same as if the transactions had been executed in a serial fashion, where one executes to completion before the other starts.

- **REPEATABLE READ**

  This setting offers the same consistency guarantee as serializable, except that the concurrency effect known as phantoms may be encountered (see below for a reference to the definition of this concurrency effect).

- **READ COMMITTED**

  This setting offers the same consistency guarantee as repeatable read, except that the concurrency effect known as non-repeatable read may also be encountered (see below for a reference to the definition of this concurrency effect).

- **READ UNCOMMITTED**

  This setting offers the same consistency guarantee as read committed, except that the concurrency effect known as dirty read may also be encountered (see below for a reference to the definition of this concurrency effect).

For a definition of the concurrency effects mentioned above (phantoms, non-repeatable read and dirty read) refer to the *Mimer SQL Reference Manual, Chapter 12, SET TRANSACTION*.

All of the isolation level settings guarantee that each transaction will be executed completely or not at all and that no updates will be lost.

Exception Diagnostics Within Transactions

The `SET TRANSACTION DIAGNOSTICS SIZE` option allows the size of the diagnostics area to be defined. An unsigned integer value specifies how many exceptions can be stacked in the diagnostics area, and examined by `GET DIAGNOSTICS`, see the *Mimer SQL Reference Manual, Chapter 12, GET DIAGNOSTICS*, in situations where repeated `RESIGNAL` operations have effectively been performed.

The `SET TRANSACTION DIAGNOSTICS SIZE` setting only affects the single next transaction to be started.

The default `SET TRANSACTION DIAGNOSTICS SIZE` setting (50 or whatever has been defined to be the default by using `SET SESSION`) applies unless an alternative is explicitly set before each transaction.

Setting Default Transaction Options

The `SET SESSION` statement can be used to define the default settings for the transaction options set by `SET TRANSACTION READ`, `SET TRANSACTION ISOLATION LEVEL` and `SET TRANSACTION DIAGNOSTICS SIZE`.

As these `SET TRANSACTION` commands only affect the single next transaction started after they are used, it is often convenient to define the desired default options for each of them.

A detailed description of the `SET SESSION` statement can be found in the *Mimer SQL Reference Manual*.

Statements in Transactions

The tables that follow summarize whether statements may or may not be used inside transactions.

Access Control Statements

Statements	Allowed	Comments
GRANT REVOKE	Yes	Must be the only statement in a transaction.

Connection Statements

Statements	Allowed	Comments
CONNECT SET CONNECTION	Yes	
DISCONNECT	Yes	A ROLLBACK is performed on any active transaction.
ENTER LEAVE (program ident)	No	

Data Definition Statements

Statements	Allowed	Comments
ALTER COMMENT CREATE DROP	Yes	Must be the only statement in a transaction.

Data Manipulation Statements

Statements	Allowed	Comments
SELECT EXPRESSION SELECT INTO FETCH INSERT DELETE DELETE CURRENT UPDATE UPDATE CURRENT	Yes	
OPEN CLOSE	Yes	ROLLBACK closes all open cursors. COMMIT closes all open non-holdable cursors.

Declarative Statements

Statements	Allowed	Comments
DECLARE CONDITION DECLARE CURSOR DECLARE HANDLER DECLARE VARIABLE	Not applicable	Declarative statement

Diagnostic Statements

Statements	Allowed	Comments
GET DIAGNOSTICS RESIGNAL SIGNAL	Yes	

Dynamic SQL Statements

Statements	Allowed	Comments
PREPARE DESCRIBE EXECUTE EXECUTE IMMEDIATE EXECUTE STATEMENT ALLOCATE CURSOR ALLOCATE DESCRIPTOR DEALLOCATE DESCRIPTOR DEALLOCATE PREPARE GET DESCRIPTOR SET DESCRIPTOR	Yes	See *Dynamic SQL* on page *60*.

ESQL Control Statements

Statements	Allowed	Comments
DECLARE SECTION WHENEVER	Not applicable	Declarative statement

Procedure Control Statements

Statements	Allowed	Comments
CALL CASE COMPOUND STATEMENT FOR IF ITERATE LEAVE LOOP REPEAT RETURN SET WHILE	Yes	

System Administration Statements

Statements	Allowed	Comments
ALTER DATABANK RESTORE ALTER DATABASE DELETE STATISTICS CREATE BACKUP SET DATABANK SET DATABASE SET SHADOW UPDATE STATISTICS	No	These statements create internal transactions to ensure data dictionary consistency

Transaction Control Statements

Statements	Allowed	Comments
SET SESSION SET TRANSACTION START	No	These statements control transaction behavior
COMMIT ROLLBACK	Yes	

Cursors in Transactions

A cursor open by the current connection may be closed implicitly by one of the transaction terminating statements `COMMIT` and `ROLLBACK`. `ROLLBACK` closes all open cursors for the current connection. `COMMIT` closes all open cursors for the current connection, except cursors declared `WITH HOLD`. Holdable cursors remain open after `COMMIT`.

When a stacked cursor is closed, all instances of the cursor are closed.

Cursors are also closed implicitly by `LEAVE` and `DISCONNECT`. In `SET TRANSACTION START EXPLICIT` mode, cursors may be opened and used outside transactions. Such cursors remain accessible when an `ENTER` statement is issued, and remain open when a `LEAVE` statement is issued.

This is illustrated in the following statement sequence:

```
...
EXEC SQL SET TRANSACTION START EXPLICIT;

EXEC SQL DECLARE c_1 CURSOR FOR SELECT col1
                                   FROM tab1;
EXEC SQL DECLARE c_2 CURSOR FOR SELECT col2
                                   FROM tab2
                                   WHERE checkcol = :VAR1;

EXEC SQL OPEN c_1;

loop
   EXEC SQL FETCH c_1
               INTO :VAR1;     -- Fetch value from tab1

   EXEC SQL ENTER ... ;        -- Change current ident

      EXEC SQL OPEN c_2;
      EXEC SQL FETCH c_2
                  INTO ...;    -- Fetch row for c_2
      EXEC SQL CLOSE c_2;

   EXEC SQL LEAVE;
end loop;
...
```

In the above example, the value fetched for the cursor `C1` is used to determine the set of rows addressed by cursor `C2`. Cursor `C1` remains open and positioned during the `ENTER … LEAVE` sequence.

Each time the loop is executed, a new value is fetched by `C1` and a new set of rows is addressed by `C2`. The same behavior applies when `LEAVE RETAIN` is used to leave a `PROGRAM` ident but keep the environment for the ident.

A cursor opened and used outside a transaction may however not be used within a transaction. If the same cursor is required outside and inside a transaction, separate instances must be opened. Remember that separate instances of a cursor address separate result sets:

```
...
EXEC SQL SET TRANSACTION START EXPLICIT;

EXEC SQL DECLARE c_1 REOPENABLE CURSOR FOR SELECT col1
                                             FROM tab1;

EXEC SQL OPEN c_1;
EXEC SQL FETCH c_1
            INTO ...;       -- First row (outside transaction)
...
EXEC SQL START;
EXEC SQL OPEN c_1;          -- New instance of cursor
EXEC SQL FETCH c_1
            INTO ...;       -- First row again
...
```

Error Handling in Transactions

In general, errors and exception conditions are reported in `SQLSTATE` after each executable SQL statement.

The value of `SQLSTATE` indicates the outcome of the preceding statement, see *SQLSTATE Return Codes* on page *323* for a list of SQLSTATE values.

`GET DIAGNOSTICS` can be used to get detailed status information after an SQL statement.

The value of `SQLSTATE` after a `COMMIT` statement indicates the success or failure of the request to commit the transaction, not the outcome of any data manipulations performed within the transaction.

About WHENEVER

Use of the general error handling statement `WHENEVER`, see the *Mimer SQL Reference Manual, Chapter 12, WHENEVER*, for a description) in transactions requires some care:

- Program control can be transferred to an exception routine in the event of an error. Make sure that the exception routine is designed to take care of uncompleted transactions.
  Most commonly, the first SQL statement in the exception routine should be `GET DIAGNOSTICS`. The exception routine should normally also execute a `ROLLBACK` statement. Remember that if the exception routine is used from a statement outside a transaction, any open cursors belonging to the current ident will be closed by the `ROLLBACK` statement. `GET DIAGNOSTICS` can be used to determine whether or not a transaction is active.
- For transaction conflict, the `SQLSTATE` value returned from the `COMMIT` statement falls into the `SQLEXCEPTION` class. If the transaction is to be retried in the event of conflict, make sure that no `WHENEVER SQLEXCEPTION GOTO exception` statement is operative.

If `WHENEVER` error handling is used in an application program, a suitable program structure for `COMMIT` statements is:

```
EXEC SQL WHENEVER SQLEXCEPTION GOTO exception;
...
EXEC SQL WHENEVER SQLEXCEPTION GOTO retry;
EXEC SQL COMMIT;
EXEC SQL WHENEVER SQLEXCEPTION GOTO exception;
...
```

Chapter 10

Distributed Transactions

Mimer SQL supports distributed transactions based on the XA interface as defined by the Open Group and Microsoft's Distributed Transaction Coordinator (DTC) protocol.

This means that Mimer SQL can be used in application environments that support distributed transactions.

Support for distributed transactions is enabled by a special key in the Mimer license file. The evaluation key that is included in the distribution enables distributed transactions. In order to use distributed transactions with other licenses obtained from Mimer, make sure that the distributed transaction option is included.

Terms and Abbreviations

In the field of distributed transactions, a number of terms and abbreviations are used. Here are the most common ones:

Term/Abbreviation	Short for	Explanation
XA	eXtended Architecture	An Open Group (X/Open) standard for distributed transaction handling. For more information, visit: http://www.opengroup.org
DTC	Microsoft Distributed Transaction Coordinator	The transaction manager used in Windows environments. For more information, visit: https://msdn.microsoft.com/library
Application Server	N/A	A program that handles all application operations between users and an organization's business applications or databases.
MTS	Microsoft Transaction Server	The Microsoft application server
EJB	Enterprise JavaBeans	The framework for application servers in the Java environment.

How Does it Work?

Normally, an application that uses distributed transactions is a component in an application server. The application server environment takes care of all transaction processing. The application component just accesses one or several database servers (Resource Managers, RM's) using normal programming interfaces.

Note that the application server or Transaction Manager (TM) are not part of the Mimer database server. These components are obtained from other sources.

When the application server environment starts a new transaction, it contacts the Transaction Manager, which assigns a new transaction id (XID) to the transaction. All operations done by the application components are automatically assigned to this XID.

When the transaction is ready to commit, the TM executes the commit operations according to the two-phase commit protocol.

In phase 1, each participating RM is asked if it is ready to prepare for commit. By replying yes, the RM promises that it is able to commit the transaction and remember everything about the transaction. Although it can not commit the transaction yet, it must secure all information on disk to make sure that no information can be lost. If the RM determines that it can not commit the transaction at a later stage, it may answer no to the TM. In this case, the TM aborts its preparation phase. It contacts all RMs again and tells them to abort the transaction.

If the TM got a yes from all RMs in the first phase, phase 2 begins. Each RM is asked to commit the transactions.

Handling failures

The Transaction Manager is responsible for performing the two-phase commit protocol. It must maintain the current state of this protocol for every transaction it manages. It should also be able to deal with failures in any component, including itself.

A problematic situation occurs if the contact between the TM and the RM is lost when the protocol is between phase 1 and 2. In this case, the RM has promised to be able to commit a certain update, but it does not yet know whether it should actually do so. This is determined in phase 2. Because of this uncertainty, the RM does not know what value to return to other transactions that asks for the information that was updated. Should the old or the new value be returned? A Mimer database server will typically abort transactions that request data which was updated by a transaction that is in doubt.

The situation is automatically resolved when the contact between the TM and the RM is reestablished. Since both TM and RM save all information on disk, they may both crash between phase 1 and 2, and still be able to carry through with the two phase commit protocol.

However, if the TM somehow fails to reconnect to a Mimer database server that has prepared transactions in doubt, there is another option. The operator may perform a **heuristic commit** or a **heuristic rollback**. By doing this, the operator does the role that the TM normally does and resolves the state of the transaction that is in doubt. This can be done by using the TRANSACTIONS command, described in *Mimer SQL User's Manual, Chapter 9, TRANSACTIONS*.

Note that if the TM has already instructed some RMs to (for example) commit, while the operator does a **heuristic rollback** on another RM, a transactional inconsistency has been introduced. This must be resolved manually. Because of this risk, heuristic operations should be used with due care.

Mimer SQL Support For Microsoft DTC on Windows

Mimer SQL supports the complete distributed transaction model according to the Microsoft Distributed Transaction Coordinator (MSDTC). This allows transactions to span several Mimer SQL databases. The database servers can be located on any type of hardware where Mimer SQL 9.1 or later is supported. To use MSDTC, the client must be on a Windows platform.

It is also possible to have transactions over heterogeneous database systems. For example, a single transaction can be performed which updates data in both a Mimer SQL database and a Microsoft SQL Server database.

The support allows code written for Microsoft Transaction Server and COM+ to be used with Mimer SQL. The transaction can be managed automatically by the COM+ server. I.e. transactional components are fully supported and transactional attributes can be 'Supported', 'Required', or 'Requires New'. Please see your COM+ documentation for a thorough discussion of these options.

There is no distributed transaction support for Mimer SQL servers older than 9.1.

Mimer SQL Support for Java Enterprise Edition

Mimer SQL has full support for the distributed transactions in Java Enterprise Edition (Java EE) through the XA support in the Mimer JDBC Driver. With Java EE any XA compliant data source can take part in a distributed transaction. The distributed transactions are handled by the Java EE Application Server and specified by the application developer in a declarative manner. This way the application developer does not have to include any transaction handling logic in the application. Distributed transactions can be used in several different parts of the Java EE framework, for example in Enterprise Java Beans (EJB).

To be able to use Mimer SQL in a distributed transaction an XA data source must be created in the application server. How this is done differ between different application servers, consult your manual to see how it is done. When defining an XA data source for Mimer SQL, the `com.mimer.jdbc.MimerXADataSource` Java class should be used.

For more information about the Mimer JDBC Driver, see *the Mimer JDBC Driver Guide*.

Chapter 11

Mimer SQL Stored Procedures

In Mimer SQL, the term stored procedures refers to routines, i.e. functions and procedures.

Mimer SQL stored procedures conform to the SQL/PSM standard. The SQL/PSM standard consists of:

- syntax and semantics for variable and cursor declarations
- assignment of the results of expressions to variables and parameters
- conditional statements
- control statements for looping and branching
- condition and exception handling
- getting diagnostics for status information and routine invocations.

Modules can be used to collect a number of routines together as a group.

Mimer SQL PSM Debugger

You can debug routines and triggers using Mimer SQL's Java-based graphic debugger for PSM routines. The debugger supports watching variables, step-wise execution and setting breakpoints. For more information, see *The Mimer SQL PSM Debugger* on page *275*.

About Routines

A routine is either defined as a function or as a procedure. Essentially the same constructs may be used in both functions and procedures.

A routine can be created by declaring it in a module definition, see *Modules* on page *253*, or be created on its own by executing the `CREATE FUNCTION` or `CREATE PROCEDURE` statement. A routine created on its own cannot be subsequently added to a module.

A routine belongs to the schema in which it was created and the routine name may be qualified in the normal way with the name of the schema. Only the ident with the same name as the schema to which a routine belongs may refer to it by its unqualified name, all other idents must use the fully qualified routine name.

It is possible to have multiple functions and procedures with the same name within a schema as long as they differ with regard to either the number of parameters or the data type for the parameters. This is called parameter overloading.

To distinguish between routines with the same name it is possible to give a specific name when creating a routine. This specific name can used when granting or revoking execute privilege for the routine or when dropping the routine.

It is possible for a function to have the same qualified name as a procedure, because the invocation of a function is distinct from that of a procedure.

In order to invoke a routine, the ident invoking it must have been granted `EXECUTE` privilege on the routine. Routines may be recursively invoked.

Note: When routines and modules are created, the create statement must be executed as one single statement. For example, using BSQL, the create statement must be delimited by the `@` character, see the *Mimer SQL User's Manual, Chapter 7, Creating Functions, Procedures, Triggers and Modules*, for details and examples.

The following points should be noted for procedures:

- they are invoked by using the `CALL` statement.
- any result from a procedure must be returned via one of the output parameters, except in the special case of a result set procedure, which can return rows of a result set to a cursor, see *Result Set Procedures* on page *264*.

The following points should be noted for functions:

- they are invoked from an SQL statement where a value is required. Certain restrictions apply, see *Invoking Functions* on page *260*. For example:

```
SET :isbn = mimer_store_book.format_isbn('1558604618');
```

- the parameters of a function provide input only and the function result is returned as the value of the function invocation.

A routine essentially consists of static SQL source that is stored in the data dictionary and which may be invoked by name whenever it is to be executed.

The SQL source for a routine comprises a definition of various routine components, see *Syntactic Components of a Routine Definition* on page *245* for details, followed by the routine body.

The routine body consists of a single executable SQL statement - typically a compound SQL statement, i.e. local declarations and a number of SQL statements delimited by a `BEGIN` and `END`. See *Scope in Routines – the Compound SQL Statement* on page *248*.

Note: It is recommended that a compound SQL statement always be used for the body of a routine, as this offers the greatest flexibility and results in a consistent structure for all routines.

It is possible to declare exception handlers within a compound SQL statement to handle specific exceptions or classes of conditions, see *Declaring Exception Handlers* on page *269*.

Functions

A function is invoked by specifying the function invocation where a value expression would normally be used. The parameters of a function are used to provide input only, values cannot be passed back to the calling environment through the parameters of a function.

A function always returns a single value and the data type of the return value is defined in the returns clause, which is specified after the parameter definition part of the function definition.

The function returns its value when a `RETURN` statement is executed within the body of the function. The data type of the value expression in the `RETURN` statement must be assignment-compatible with the data type specified in the returns clause of the function.

Functions and SQL Statements

The SQL statements that apply to a function are:

Statement	Description
`ALTER FUNCTION`	alters an already existing function, see *Mimer SQL Reference Manual, Chapter 12, ALTER FUNCTION*
`CREATE FUNCTION`	creates a function that exists on its own, see the *Mimer SQL Reference Manual, Chapter 12, CREATE FUNCTION*
`DROP FUNCTION`	drops a function that exists on its own, see the *Mimer SQL Reference Manual, Chapter 12, DROP*
`GRANT EXECUTE`	grants the privilege to invoke a function, see the *Mimer SQL Reference Manual, Chapter 12, GRANT OBJECT PRIVILEGE*
`REVOKE EXECUTE`	revokes the privilege to invoke a function, see the *Mimer SQL Reference Manual, Chapter 12, REVOKE OBJECT PRIVILEGE*
`COMMENT ON FUNCTION`	defines a comment on a function, see the *Mimer SQL Reference Manual, Chapter 12, COMMENT*.

Example 1

```
CREATE FUNCTION SQUARE_INTEGER(p_root INTEGER) RETURNS INTEGER
CONTAINS SQL
BEGIN
   RETURN p_root * p_root;
END
```

Example 2

```
CREATE FUNCTION mimer_store_web.session_expiration_period()
   RETURNS INTERVAL HOUR TO MINUTE
-- Defines the period that a session can be unused
DETERMINISTIC
RETURN INTERVAL '10' MINUTE(3);  -- Intentionally very short
```

Example 3

```
CREATE FUNCTION date_plus_time (d date, t time(6))
  RETURNS timestamp
-- Create a timestamp, from a date plus time input
DETERMINISTIC
RETURN cast(d as timestamp) + (t - time '00:00:00') hour to second(6);
```

Example 4

```
CREATE FUNCTION mimer_store_book.keyword_id(p_keyword VARCHAR(48))
   RETURNS INTEGER
-- Inserts a word in the KEYWORDS table
-- and returns the identifier with which the keyword is associated
MODIFIES SQL DATA
BEGIN
   DECLARE v_keyword_id INTEGER;

   DECLARE CONTINUE HANDLER FOR NOT FOUND
   BEGIN
      INSERT INTO mimer_store_book.keywords(keyword)
         VALUES (UPPER(TRIM(p_keyword)));

      SET v_keyword_id = CURRENT VALUE FOR mimer_store_book.keyword_id_seq;
   END;  -- of not found handler

   SELECT keyword_id
      INTO v_keyword_id
      FROM mimer_store_book.keywords
      WHERE keyword = TRIM(p_keyword);

   RETURN v_keyword_id;
END  -- of routine mimer_store_book.keyword_id
```

Procedures

A procedure is normally invoked explicitly by executing the `CALL` statement and does not return a value. The parameters of a procedure can be used to provide input and may be used to pass values back to the calling environment.

There is a special type of procedure, called a result set procedure, which returns rows of a result set to a cursor when it is invoked by executing the `FETCH` statement in that context.

A result set procedure is distinguished from a normal procedure by having a returns clause specified after the parameter definition part of the procedure definition, see *Result Set Procedures* on page *264* for a detailed description of result set procedures.

Procedures and SQL Statements

The SQL statements that apply to a procedure are:

Statement	Description
`ALTER PROCEDURE`	alters an already existing procedure, see *Mimer SQL Reference Manual, Chapter 12, ALTER PROCEDURE*
`CREATE PROCEDURE`	creates a procedure that exists on its own, see the *Mimer SQL Reference Manual, Chapter 12, CREATE PROCEDURE*
`DROP PROCEDURE`	drops a procedure that exists on its own, see the *Mimer SQL Reference Manual, Chapter 12, DROP*
`GRANT EXECUTE`	grants the privilege to invoke a procedure, see the *Mimer SQL Reference Manual, Chapter 12, GRANT OBJECT PRIVILEGE*
`REVOKE EXECUTE`	revokes the privilege to invoke a procedure, see the *Mimer SQL Reference Manual, Chapter 12, REVOKE OBJECT PRIVILEGE*

Statement	Description
CALL	invokes a procedure, see the *Mimer SQL Reference Manual, Chapter 12, CALL*
COMMENT ON PROCEDURE	defines a comment on a procedure, see the *Mimer SQL Reference Manual, Chapter 12, COMMENT*.

Example 1

```
CREATE PROCEDURE mimer_store_web.delete_basket(p_session_no VARCHAR(16))
-- Deletes expired baskets
MODIFIES SQL DATA
BEGIN
   IF p_session_no = '*' THEN
      -- '*' indicates that all expired sessions should be deleted
      DELETE
         FROM mimer_store.orders
         WHERE order_id IN (SELECT order_id
                            FROM mimer_store_web.sessions
                            WHERE last_accessed < LOCALTIMESTAMP -
                            mimer_store_web.session_expiration_period());
   ELSE
      -- Delete the specified session
      DELETE
         FROM mimer_store.orders
         WHERE order_id = (SELECT order_id
                           FROM mimer_store_web.sessions
                           WHERE session_no = p_session_no);
   END IF;
END  -- of routine mimer_store_web.delete_basket


CALL mimer_store_web.delete_basket( '*' );


COMMENT ON PROCEDURE mimer_store_web.delete_basket
        IS 'Deletes expired baskets';


DROP PROCEDURE mimer_store_web.delete_basket;
```

Example 2

```
CREATE PROCEDURE mimer_store_book.catalogue_authors(IN p_item_id INTEGER,
                                                    IN p_authors_list VARCHAR(128))
-- Stores author names as keywords and forms a link between a book
-- and the keywords
MODIFIES SQL DATA
BEGIN
   DECLARE v_author VARCHAR(50);
   DECLARE v_authors VARCHAR(130);
   DECLARE v_offset, v_length INTEGER;

   SET v_authors = REPLACE(' ' || p_authors_list || ' ', ' and ', ';');
   SET v_authors = REPLACE(v_authors, ' & ', ';');
   SET v_authors = TRIM(v_authors);

extract_authors:
   LOOP
      IF v_authors = '' THEN LEAVE extract_authors; END IF;

      SET v_offset = POSITION(';' IN v_authors);

      IF v_offset <> 1 THEN
         IF v_offset = 0
         OR v_offset > 49 THEN
            SET v_length = 48;
         ELSE
            SET v_length = v_offset - 1;
         END IF;

         SET v_author = mimer_store_book.authors_name(
                           SUBSTRING(v_authors FROM 1 FOR v_length));

         BEGIN
            DECLARE v_keyword_id INTEGER;

            DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
            BEGIN
               -- Ignore all SQL errors
            END;  -- of sqlexception handler

            SET v_keyword_id = mimer_store_book.keyword_id(v_author);

            INSERT INTO mimer_store_book.authors(keyword_id, item_id)
               VALUES (v_keyword_id, p_item_id);
         END;
      END IF;

      IF v_offset = 0 THEN LEAVE extract_authors; END IF;

      SET v_authors = TRIM(SUBSTRING(v_authors FROM v_offset+1));
   END LOOP extract_authors;
END  -- of routine mimer_store_book.catalogue_authors
```

Syntactic Components of a Routine Definition

The following sections discuss parameters, language indicators, clauses, scope, variables and data types when working with routines.

Routine Parameters

A routine may have zero or more parameters and each parameter must have a name and a data type specified.

Each parameter of a procedure can have an optional mode specification (`IN`, `OUT` or `INOUT` – see `CREATE PROCEDURE` in the *Mimer SQL Reference Manual* for details). When the mode is not explicitly specified, `IN` is assumed by default.

It is not possible to specify the mode for the parameters of a function (they always have the default mode, `IN`).

A parameter name must be unique within the routine. The parameter name can be up to 128 characters in length, see the *Mimer SQL Reference Manual, Chapter 6, Naming Objects*, for further details about naming SQL objects.

The parameter data type can be any data type supported by Mimer SQL, except any of the large object types, see the *Mimer SQL Reference Manual, Chapter 6, Data Types in SQL Statements*.

A parameter name may be referenced in an unqualified manner throughout a routine, at all scope levels – see *Scope in Routines – the Compound SQL Statement* on page *248* for a discussion of scope in routines.

Examples:

```
CREATE FUNCTION onefunction(a INTEGER, b DECIMAL(5,2))
RETURNS DECIMAL(5,2)
BEGIN
   ...
END

CREATE PROCEDURE lookup(IN i INTEGER, OUT retval VARCHAR(20))
BEGIN
   ...
END
```

Parameter Overloading

Mimer SQL supports the possibility to define multiple functions or procedures with the same name as long as they differ with regard to the number of parameters or the data type of the parameter. It is not possible to have multiple functions that only differ with regard to the return data type.

As an example, it is possible to create two functions like

```
SQL>create function f(c1 char) returns int return 1;
SQL>create function f(c1 integer) returns int return 2;
```

and these can be used as

```
SQL>set ? = f('1');
           ?
===========
           1


SQL>set ? = f(1);
           ?
===========
           2
```

In this case there is no problem deciding which routine that should be invoked in the two cases since there can be no implicit conversion from a character type to integer or vice versa. Some interesting cases arise when there are parameter overloading and where there are implicit conversions between the parameter types. For instance, if we have these functions

```
SQL>create function f(p char varying(2))
SQL&  returns int return 1;
SQL>create function f(p nchar varying(2))
SQL&  returns int return 2;
```

Given the statement

```
SQL>set ? = f('a');
```

which routine should be invoked? The data type of the actual argument is char and there is no routine with a parameter list that matches the invocation exactly. In this case a type precedence list is used to determine the proper subject routine. For `CHAR` the type precedence list is `CHAR`, `CHARACTER VARYING`, `NCHAR` and `NCHAR VARYING` which means that the function returning 1 will be chosen in this case. If there are multiple parameters, the subject routine is determined by evaluating the type precedence list for each parameter, going from left to right.

The type precedence lists are found in *Mimer SQL Reference Manual, Appendix H, Type Precedence Lists*.

Specific Names and Parameter List

When specifying a routine in a drop, grant or revoke statement the routine must be uniquely identifiable. This is no problem as long as parameter overloading is not used. If there are multiple functions or procedures with the same name there are two ways of specifying a unique routine. The first is by using the specific name for the routine. A specific name can be defined when the routine is created. If no specific name is given, a unique name is generated automatically. This name can be seen in the view `INFORMATION_SCHEMA.ROUTINES`. As an example:

```
CREATE PROCEDURE p(p1 INT, p2 CHAR(20))
SPECIFIC p_int_char
…
GRANT EXECUTE ON SPECIFIC PROCEDURE p_int_char
TO public WITH GRANT OPTION;
```

The other way to distinguish between overloaded routines in DDL statements is to use a data type list. Given the above procedure definition, the grant statement can be written as

```
GRANT EXECUTE ON PROCEDURE p(INT,CHAR) TO public WITH GRANT OPTION;
```

To specify a routine without parameters, the syntax is

```
GRANT EXECUTE ON PROCEDURE p() TO public WITH GRANT OPTION;
```

Routine Parameters and Null Values

All parameters accept null values. Use `CAST` to invoke a routine with a null value as a parameter, as follows:

```
CALL mimer_store_music.AddTrack(718751799622, CAST(NULL AS int), 'Null Set',
  '3:53')
```

Routine Language Indicator

The language indicator specifies the language of the routine. Currently, the only language name supported is SQL.

If no language indicator is specified, `LANGUAGE SQL` is assumed by default.

Routine Deterministic Clause

The deterministic clause for a routine can specify `NOT DETERMINISTIC` or `DETERMINISTIC`. If a deterministic clause is not specified, `NOT DETERMINISTIC` is assumed by default.

A `DETERMINISTIC` routine is one that is guaranteed to produce the same result every time it is invoked with the same set of input values.

Therefore, a `DETERMINISTIC` routine may not contain a reference to: `CURRENT_DATE`, `LOCALTIME`, `LOCALTIMESTAMP` or `BUILTIN.UTC_TIMESTAMP`.

Specifying a routine to be `DETERMINISTIC` allows repeated invocations of it to be optimized.

Routine Access Clause

The access clause for a routine specifies which SQL statements are permitted within the routine.

The three different options for the routine access clause (`CONTAINS SQL`, `READS SQL DATA` and `MODIFIES SQL DATA`) are described under `CREATE PROCEDURE` in the *Mimer SQL Reference Manual*. If no routine access clause is specified, then `CONTAINS SQL` is assumed.

If the routine contains a `SELECT` statement, `READS SQL DATA` is required. (Or if a `READS SQL DATA` routine is called.)

If the routine contains a `DELETE` or an `UPDATE` statement, `MODIFIES SQL DATA` is required. (Or if a `MODIFIES SQL DATA` routine is called.)

Scope in Routines – the Compound SQL Statement

A compound SQL statement allows a sequence of procedural SQL statements to be considered as a single SQL statement, see COMPOUND STATEMENT in the *Mimer SQL Reference Manual* for a description of the syntax.

A routine body may contain only one executable SQL statement and the compound SQL statement allows a routine to be defined that can actually contain any number of SQL statements.

A compound SQL statement also defines a local scope in which variables, exception handlers, and cursors can be declared. Compound SQL statements may be nested, one within the other, and thus local scopes may be nested.

A compound SQL statement may be labeled, which effectively names the local scope defined by it. The label name can be used whenever the scope environment needs to be referred to explicitly, e.g. when qualifying the names of objects which have been declared in the compound SQL statement. The label name must not be the same as a routine name.

It is important to understand the effect of scoping on declared items, particularly with respect to: out-of-scope references to variables, see *Declaring Variables* on page *250*, the scope within which an exception handler remains in effect and the flow of control effects following the use of different types of exception handler, see *Declaring Exception Handlers* on page *269*.

The SQL statement LEAVE is specifically provided to give the programmer the ability to force the flow of control to exit from a labeled scope.

Example

```
CREATE PROCEDURE some_procedure(INOUT y INTEGER)
CONTAINS SQL
s0:
BEGIN
   ...
s1:
   BEGIN
      IF y < 0 THEN
         SET y = 0;
         LEAVE s0;
      END IF;
      ...
   END s1;
   ...
END s0
```

In the example above, the effect of the LEAVE statement is to pass flow of control to the statement END s0, i.e. flow of control exits from the scope labeled s0.

All open cursors declared in a compound SQL statement are closed whenever flow of control leaves the compound SQL statement for any reason.

Note: A compound SQL statement may be preceded by a label which names the scope delimited by the BEGIN and END (this is called the beginning label). Specifying the label next to the END is optional. However, if a label is specified next to the END, the beginning label must be specified.

The ATOMIC Compound SQL Statement

The execution of any SQL statement, other than a `procedure-control-statement`, is atomic. See the *Mimer SQL Reference Manual, Chapter 12, Procedural SQL Statements*, for a definition of a `procedure-control-statement`.

The execution of a compound SQL statement defined as ATOMIC is also atomic.

When the execution of an SQL statement is atomic, an atomic execution context becomes active while the statement, or any contained subquery, is executing. While an atomic execution context is active, it is possible for another atomic execution context to become active within it.

While an atomic execution context is active the following is true:

- It is not possible to explicitly terminate a transaction, thus all changes made within the atomic execution context occur within the same transaction.
- If an exception occurs during the execution of a statement and there is an undo handler declared for this exception, then all delete, insert and update statements executed within the atomic compound statement are undone. If there is no undo handler, only the statement that caused the exception will be undone.

Note: If the atomic statement contains operations on tables located in a databank defined with work option, these operations will not be part of the atomic statement but will be executed immediately. If the atomic statement is terminated by an SQL exception, such operation will not be undone.

An atomic compound SQL statement is defined by specifying the keyword `ATOMIC` next to the `BEGIN` delimiter. The `COMMIT` and `ROLLBACK` statements cannot be used within an atomic compound SQL statement.

A compound SQL statement is explicitly defined as not being atomic by specifying `NOT ATOMIC` next to the `BEGIN` delimiter. If nothing is specified next to the `BEGIN` delimiter, `NOT ATOMIC` is assumed by default.

If the compound SQL statement contains a declaration for an `UNDO` exception handler, see *Declaring Exception Handlers* on page *269*, the compound SQL statement must be `ATOMIC`.

Examples:

```
CREATE FUNCTION an_atomic_function(i INTEGER)
RETURNS INTEGER
BEGIN ATOMIC
   ...
   -- All statements executed between this BEGIN
   -- and END execute within the same active atomic
   -- execution context.
   -- UNDO exception handlers are permitted.
   -- No COMMIT or ROLLBACK allowed!
   ...
END

CREATE PROCEDURE a_non_atomic_procedure(i INTEGER)
BEGIN NOT ATOMIC
   ...
   -- This compound SQL statement is not atomic.
   -- COMMIT and ROLLBACK statements are permitted.
   -- No UNDO exception handlers allowed!
   ...
END
```

```
CREATE FUNCTION a_default_function(i INTEGER) RETURNS INTEGER
BEGIN
   ...
   -- This compound SQL statement is not atomic, by default.
   -- COMMIT and ROLLBACK statements are permitted.
   -- No UNDO exception handlers allowed!
   ...
END
```

Declaring Variables

It is possible to declare variables, cursors, condition names and exception handlers at the beginning of a compound SQL statement. These items can, therefore, be declared in a routine when a compound SQL statement is used for the routine body.

This section discusses the declaration of variables. Discussions about declaring the other items mentioned above can be found in the following sections:

- cursors, see *Using Cursors* on page *262*
- condition names, see *Declaring Condition Names* on page *268*
- exception handlers, see *Declaring Exception Handlers* on page *269*.

Variables of any data type supported by Mimer SQL may be declared. The name of a variable must be unique within the scope of its declaration and must not conflict with the name of any of the routine parameters.

Variable names can be a maximum of 128 characters in length and are case insensitive. See the *Mimer SQL Reference Manual, Chapter 6, Naming Objects*, for further details.

More than one variable of the same type can be declared in a single variable declaration, see the examples below.

It is possible to specify an optional expression, which may be null, that defines the default value for a variable declaration. The variable(s) created by the variable declaration are given the initial value derived from the default expression. If a default expression is not specified, the value null is assumed.

Examples:

```
DECLARE z INTEGER;
DECLARE x, y INTEGER DEFAULT 9;
DECLARE abx VARCHAR(50);
DECLARE a INTEGER DEFAULT NULL;
```

Note: It is possible to declare a variable that has the same name as a column name in a table. All ambiguous references will be interpreted as a reference to a column name rather than a variable. It is therefore recommended that a suitable naming convention be adhered to that clearly distinguishes between the names of table columns and variables.

The name of a variable may be qualified in the normal way with the beginning label of the scope in which it has been declared.

Example

```
CREATE PROCEDURE some_procedure(IN x INTEGER)
s0:
BEGIN
   DECLARE a, b INTEGER;
s1:
   BEGIN
      DECLARE b, c INTEGER;
      ...
   END s1;
s2:
   BEGIN
      DECLARE y INTEGER;
      ...
   END s2;
END s0
```

The qualified names for the variables in the preceding example are as follows:

`s0.a`, `s0.b`, `s1.b`, `s1.c` and `s2.y`.

The ROW Data Type

Mimer SQL supports a data type called the `ROW` data type. It can be used in a compound SQL statement in place of the data type specified when a variable is declared.

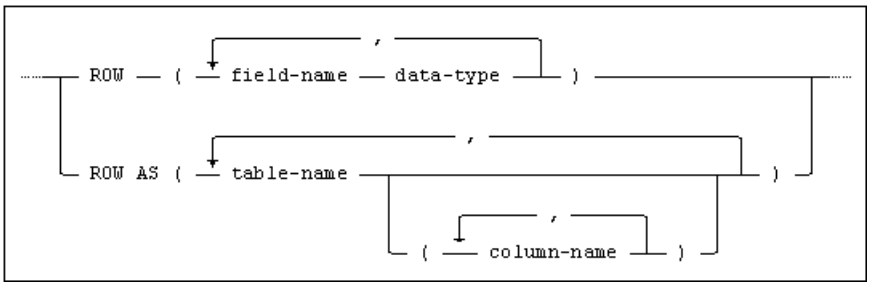
A variable that is declared as having the `ROW` data type implicitly defines a row value, which is a single construct that has a value that effectively represents a table row.

A row value is composed of a number of named values, each of which has its own data type and represents a column value in the overall row value.

A `ROW` data type can be defined either by explicitly specifying a number of field-name/data-type pairs or by specifying a number of table columns from which the unqualified names and data types are inherited.

ROW Data Type Syntax

The syntax for defining a `ROW` data type is:



The following points apply to the specification of a `ROW` data type:

- A domain cannot be specified for `data-type`.
- The value specified for `data-type` can be a `ROW` data type specification.
- Two fields in the same `ROW` data type specification must not have the same name (this restriction applies equally to fields named by specifying a `field-name` value and those named by inheriting the unqualified name of a table column).
- If `table-name` is specified without a list of column names, all the columns in the table are used to define fields in the `ROW` data type.

Using the ROW Data Type

A ROW variable field is referenced like this: variable-name.field-name.

A value may be assigned to one of the fields in a ROW variable in the same way as a value would be assigned to a variable declared with the same data type as the field. The data type of the field must be assignment compatible with the value being assigned to it.

If the declaration of a ROW variable does not include a DEFAULT clause, each field in the ROW variable is set to null initially.

The value of a field in a ROW variable may be used in the same way as any value of that type.

When a ROW data type is defined by specifying table columns, the names and data types of its fields are inherited from the columns in the table(s). Subsequently assigning values to the ROW variable will not affect the table(s) used to define the ROW data type.

A row value, which may be the value of a ROW variable, may be assigned to a ROW variable. The row value and the ROW variable are assignment-compatible if, and only if, both contain the same number of values and each value in the row value is assignment-compatible with the corresponding field in the ROW variable.

Two row values, one or both of which may be the value of a ROW variable, may be compared. The row values are comparison-compatible if, and only if, both contain the same number of values and each value in one is comparison-compatible with the corresponding value in the other.

A ROW variable may be used within a compound SQL statement in the following contexts:

- As the only expression specified in a RETURN statement used in a result set procedure. The ROW variable must be assignment-compatible with the row value defined by the procedure VALUES clause.
- As the only target variable specified in the INTO clause of a SELECT INTO statement. The row value selected must be assignment-compatible with the ROW variable and will be assigned to it.
- As the only target variable specified in the INTO clause of a FETCH statement. The row value fetched must be assignment-compatible with the ROW variable and will be assigned to it.
- As the procedure-variable or expression in a SET assignment statement (see the description above of assignment-compatibility involving ROW variables).
- As an argument in a comparison (see the description above of comparison-compatibility involving ROW variables).

Row Value Expression

A row value expression is an expression that specifies a row value. The values that represent the column values of the row value expression are specified as value expressions in a comma-separated list that is delimited by parentheses.

A row value expression can be used in the following contexts:

- As the only expression in a RETURN statement used in a result set procedure.
- As the expression following the DEFAULT keyword in a DECLARE VARIABLE statement for a variable declared to have the ROW data type.
- As a row value in a comparison or assignment operation.

Examples:

```
RETURN (24, 16, 'xyz', 11.3, x+4/9);

DECLARE rc ROW (a INTEGER, b INTEGER, s VARCHAR(10))
   DEFAULT (14, 27, 'hello');

IF rc = (14, 27, 'hello') THEN
   SET rc.s = 'bye';
END IF;

SET rc = (99, 105, 'new value');
```

Modules

A module is a collection of routines. All the routines in a module are created by declaring them when the module is created. Routines cannot be added to or removed from a module after the module has been created.

A module belongs to the schema in which it is created and all the routines contained in a module must belong to the same schema as the module.

The name of a routine in a module may be qualified in the normal way by using the name of the schema to which the routine belongs. The module name is never used to qualify the name of a routine.

Note: It is not possible to grant `EXECUTE` privilege on a module. In order to allow an ident to invoke a routine, whether it exists on its own or in a module, `EXECUTE` privilege on the routine must be granted to the ident.

When a module is dropped, all the routines in the module will be dropped as well. See *Using DROP and REVOKE* on page *275* for a discussion of `CASCADE` effects on modules and routines.

The operations that may be performed on a module are:

- `CREATE MODULE`
- `DROP MODULE`
- `COMMENT ON MODULE`

Refer to the *Mimer SQL Reference Manual, Chapter 12, SQL Statements* for a description of the SQL statements mentioned above, brief examples follow.

Examples:

```
CREATE MODULE module_1
   DECLARE PROCEDURE p1 ... ;
   DECLARE PROCEDURE p2 ... ;
   DECLARE FUNCTION f1 ... ;
   ...
END MODULE

COMMENT ON MODULE module_1 IS 'This is my example module';

DROP MODULE module_1 CASCADE;
```

SQL Constructs in Routines

The following SQL constructs are specifically provided for use in the body of a routine.

Assignment Using SET

The SET statement is used to assign a value to a variable declared in a routine or an output parameter of a procedure (i.e. a parameter with mode OUT or INOUT).

Examples:

```
SET a = 5;
SET x = NULL;
SET y = 11 + a;
SET d = CURRENT_DATE;
SET z = NEXT VALUE FOR Z_SEQUENCE;
SET (x, y) = (CASE y WHEN 1 THEN y ELSE 0 END, 64);
```

Conditional Execution Using IF

The IF statement provides a mechanism for conditional execution of SQL statements based on the truth value of a conditional expression.

Note: If the conditional expression includes (or equals) null, the conditional expression evaluates to false. Testing for the null value must be done by using IS NULL, see the *Mimer SQL Reference Manual, Chapter 9, The NULL Predicate*.

A basic IF statement consists of a conditional expression followed by a list of one or more SQL statements in a THEN clause, which are executed if the conditional expression evaluates to true and, optionally, a list of one or more SQL statements in an ELSE clause which are executed if the conditional expression evaluates to false.

All of the predicates supported by Mimer SQL are allowed in the conditional expression of an IF statement – see the *Mimer SQL Reference Manual, Chapter 9, Predicates*.

One or more IF statements can be nested, one within the other, by using an ELSEIF clause in place of the ELSE clause in the IF statement containing another.

The IF statement does not in any sense define a local scope, it is simply a mechanism for conditionally executing a sequence of SQL statements.

Once the SQL statements to be executed have been selected, they execute in the same way as any ordinary sequence of SQL statements. This point is particularly important when considering exception condition handling behavior, see *Managing Exception Conditions* on page *267*.

Examples:

```
IF x > 50 THEN
   SET x = 50;
   SET y = 1;
ELSE
   SET y = 0;
END IF;

IF y IN (2,3,4) THEN
   ...
ELSE
   ...
END IF;

IF x > 50 THEN
   SET x = 50;
   SET y = 2;
ELSEIF x > 25 THEN
   SET y = 1;
ELSE
   SET y = 0;
END IF;

IF NOT EXISTS (SELECT *
                  FROM table_1) THEN
   ...
ELSE
   ...
END IF;

IF X > (SELECT c1
           FROM t1
           WHERE ... ) THEN
   ...
ELSE
   ...
END IF;
```

Conditional Execution – the CASE Statement

The CASE statement provides another mechanism for conditional execution of SQL statements. The CASE statement comes in two forms, a simple case and a searched case.

Simple Case

A simple case works by evaluating equality between one value expression and one or more alternatives of a second value expression. For example:

```
DECLARE y INTEGER;

CASE y
   WHEN 1 THEN ...
   WHEN 2 THEN ...
   WHEN 3 THEN ...
   ELSE ...
END CASE;
```

Searched Case

A searched case works by evaluating, for truth, a number of alternative search conditions. For example:

```
CASE
   WHEN EXISTS (SELECT *
                  FROM BILL) THEN ...
   WHEN x > 0 OR y = 1 THEN ...
   ELSE ...
END CASE;
```

About Case Statements

For both forms of the CASE statement the following is true:

- A sequence of one or more SQL statements can follow the THEN clause for each of the conditional alternatives, in the same way as for an IF statement, even though only a single implied SQL statement is shown in the examples above.
- Each alternative sequence of SQL statements in a CASE statement is treated in the same way, with respect to the behavior of exception handlers etc., as has already been described for sequences of SQL statements in an IF statement, see *Conditional Execution Using IF* on page *254*.
- Like the IF statement, the CASE statement simply provides a mechanism for selecting a sequence of SQL statements to execute. The CASE statement as a whole is not considered, in any sense, to be a single statement.
- The conditional part of each WHEN clause is evaluated, working from the top of the CASE statement down. The SQL statements that are actually executed are those following the THEN clause of the first WHEN condition to evaluate to true. If none of the WHEN conditions evaluate to true, the SQL statements following the CASE statement ELSE clause are executed.

The presence of an ELSE clause in the CASE statement is optional and if it is not present (and none of the WHEN conditions evaluate to true) an exception condition is raised to indicate that a case was not found for the CASE statement.

Note: If it is desired that there be no operation performed and no exception condition raised if none of the WHEN conditions evaluate to true, then an ELSE clause should be specified as an empty compound SQL statement.

Only the single selected sequence of SQL statements that follow a THEN or the ELSE is executed before the CASE statement terminates. There is no potential fall-through to subsequent THEN sequences as is found in case statements in some other programming environments.

Note: The **CASE statement** is distinct from the **CASE expression** – see the *Mimer SQL Reference Manual, Chapter 12, CASE* and *Mimer SQL Reference Manual, Chapter 9, CASE Expression*.

Iteration

The following sections describe how you can use iteration.

Iterating through a result set - FOR loop

A for loop can be used to iterate through all records in a result set and perform some operations for each record. This is a vast simplification compared to using a cursor.

A simple example of a for loop is

```
FOR SELECT surname, forname FROM customers
    WHERE customer_id IN
    (SELECT customer_id FROM orders
     WHERE datetime BETWEEN DATE '2006-01-01' AND DATE '2006-06-31') DO
     CALL orderStat(surname,forname);
END FOR
```

I.e. call the `orderStat` routine for each record in the customers table that fulfil the where criteria. Within the body of the for statement it is possible to reference the column values as ordinary variables. This also means that each item in the select list must have a name and that name must be unique within the select list.

The body of the for statement is an atomic statement, which means that it cannot contain statements such as start, commit and rollback.

It is possible to use a result set procedure in a for loop

```
FOR CALL coming_soon('Blues') DO
    IF producer IN ('Bill Vernon','Bill Ham') THEN
       INSERT INTO stats(format,release_date,…)
           VALUES (format,release_date,…);
    END IF;
END FOR
```

In this case the correlation names in the returns clause of the result set procedure definition can be used as variable names in the body of the for loop.

The select or call statement in the for loop can be labelled and this label can be used to qualify variable references.

```
l1: BEGIN
    DECLARE forname CHAR(12);
    …
    FOR l2 AS SELECT forname FROM customers DO
        IF l1.forname <> l2.forname THEN
            …
        END IF;
    END FOR;
END
```

The label used cannot be the same as any label of a compound statement enclosing the for loop.

Iteration Using LOOP

The `LOOP` statement may be preceded by a label that can be used as an argument to `LEAVE` in order to terminate the loop. The `LOOP` statement can contain a sequence of one or more SQL statements that are executed, in order, repeatedly.

The iteration is terminated by executing the `LEAVE` statement, or if an exception condition is raised.

Example

```
s1:
LOOP
...
    IF ecounter > 10000 THEN
        LEAVE s1;
    END IF;
END LOOP s1;
```

Iteration Using WHILE

The WHILE statement may be preceded by a label that can be used as an argument to LEAVE in order to terminate the while loop. The WHILE statement can contain a sequence of one or more SQL statements that are executed, in order, repeatedly.

The WHILE statement includes a conditional expression and iteration continues as long as this expression evaluates to true. Iteration may also be terminated by executing the LEAVE statement, or if an exception condition is raised.

Example

```
SET i = 0;
s1:
WHILE i <= 10 DO
   ...
   SET i = i + 1;
END WHILE s1;
```

Iteration Using REPEAT

The REPEAT statement may be preceded by a label that can be used as an argument to LEAVE in order to terminate the repeat loop. The REPEAT statement can contain a sequence of one or more SQL statements which are executed, in order, repeatedly.

The REPEAT statement includes an UNTIL clause, which specifies a conditional expression, and iteration continues until this expression evaluates to true. Iteration may also be terminated by executing the LEAVE statement, or if an exception condition is raised.

Example

```
SET i = 0;
s1:
REPEAT
   ...
   SET i = i + 1;
UNTIL i > 10
END REPEAT s1;
```

Using ITERATE to Skip Statements

You can use an ITERATE statement to skip the remaining statements in an iteration as shown in the following examples:

```
SET x = 0:
s1:
REPEAT
   SET x = x + 1;
   ...
   IF x < 10 THEN
      ITERATE s1; -- execution continues at the beginning
                  -- of the repeat statement
   END IF;
   ...
UNTIL x = 20 END REPEAT s1;
```

Using ITERATE in all Iteration Statements

You can use ITERATE in all iteration statements in stored procedures. ITERATE is not restricted to the innermost statement. For example:

```
   SET x = 0;
s1:
   REPEAT
      SET x = x + 1;
s2:
      BEGIN
s3:
         LOOP
            ...
            IF x < 10 THEN
               ITERATE s1;
            ELSEIF x < 20 THEN
               ITERATE s3;
            END IF;
            ...
         END LOOP s3;
      END s2;
   UNTIL x = 20
   END REPEAT s1;
```

Note: The statement `ITERATE s1` will cause an implicit leave of the compound statement labeled `s2`.

Invoking Procedures and Functions

The following sections discuss invoking procedures and functions.

Invoking Procedures – CALL

The CALL statement is used to invoke a procedure. The name of the procedure may be qualified with the name of the schema to which it belongs. A value expression or target variable must be specified for each of the procedure's parameters, see the *Mimer SQL Reference Manual, Chapter 6, Target Variables*, for the definition.

If the procedure parameter has mode OUT or INOUT, a target variable must be specified. For procedure parameters with mode IN, a value expression may be specified.

SQL/PSM is not strongly typed, so the expression specified for each procedure parameter need not have exactly the same data type as the parameter, however the expression must be assignment-compatible with the procedure parameter for which it is supplied, see the *Mimer SQL Reference Manual, Chapter 7, Assignments*, for a discussion of assignment and implicit data type conversions.

Examples:

```
CALL PROC1( );

CALL PROC2(x, y);

CALL IDENT1.PROC7(CURRENT_DATE, x+3, z);
```

Invoking Functions

Functions are not invoked by calling them explicitly. A function is invoked, and it returns its value, when it is used in a `procedure-control-statement` or in an assignment where a `value-expression` would normally be used.

The name of the function may be qualified with the name of the schema to which it belongs.

If `MODIFIES SQL DATA` has been specified for the `access-clause` of the function, it must not be used in the expression following the `DEFAULT` keyword in a `DECLARE VARIABLE` statement.

Examples:

```
IF fn(x) > 70 THEN
   ...
ELSE
   ...
END IF;

SET v_Artist = Mimer_Store_Music.ArtistName(p_RecordedBy) || '%';

IF Mimer_Store.Index_Text(Data.Title) LIKE v_Title THEN
   ...
END IF;
```

Comments in Routines

Any text that occurs after `--` and before end-of-line in a routine is taken to be a comment.

Example

```
CREATE PROCEDURE tstproc(y INTEGER)
-- This is a comment: Note that Y has mode IN (default)
READS SQL DATA
BEGIN
   DECLARE b INTEGER;
   -- Here is another comment
   SET b = y + 22;  -- Y is input to the procedure
   ...
END
```

Restrictions

The following groups of SQL statements may not be used in a routine:

- Access Control statements
- Data Definition statements
- Connection statements
- ESQL Control statements
- Security Control statements

- Dynamic SQL statements
- System Administration statements.

Refer to the *Mimer SQL Reference Manual, Chapter 12, SQL Statements* for a definition of the statement groups mentioned above.

Note: Any SQL statements used in a routine must be executable, so the usual restriction on the use of `SELECT` versus `SELECT INTO` applies (only the latter being considered executable - the former may, however, be used in a conditional expression, e.g. in an `IF` statement or a cursor declaration).

The following restrictions apply to result set procedures:

- A `COMMIT` or `ROLLBACK` statement must not be executed in a result set procedure because it will interfere with the open cursor that will exist in the context from where the result set procedure is called.
- A function or procedure that executes a `COMMIT` or `ROLLBACK` statement must not be invoked from within a result set procedure.
- A function or procedure that has `MODIFIES SQL DATA` specified for its access clause must not be invoked from within a result set procedure.

Manipulating Data

The following sections discuss how to use write operations, cursors and `SELECT INTO` when manipulating data.

Write Operations

You can use `INSERT`, `UPDATE` and `DELETE` statements in a function or procedure provided `MODIFIES SQL DATA` has been specified for the access clause, see *Routine Access Clause* on page *247*.

You can use routine parameters and variables in these statements wherever an expression can normally be used, as shown in the examples below.

Example

```
CREATE PROCEDURE mimer_store_book.add_title(IN p_book_title VARCHAR(48),
                                            IN p_authors VARCHAR(128),
                                            IN p_published_by VARCHAR(48),
                                            IN p_format VARCHAR(20),
                                            IN p_isbn CHAR(18),
                                            IN p_date_released CHAR(10),
                                            IN p_price DECIMAL(7, 2),
                                            IN p_stock SMALLINT,
                                            IN p_reorder_level SMALLINT)
-- Add the details for a book entity; inserts against the join view which fires
-- the instead of trigger
MODIFIES SQL DATA
BEGIN
   -- Insert into join view
   INSERT INTO mimer_store_book.details(title, authors_list, publisher,
                                        format,isbn, release_date,
                                        price, stock, reorder_level)
      VALUES (p_book_title, p_authors, p_published_by, p_format,
              p_isbn, p_date_released, p_price, p_stock,
              p_reorder_level);
END  -- of routine mimer_store_book.add_title
```

ROW_COUNT Option

You can use the ROW_COUNT option of the GET DIAGNOSTICS statement may be used immediately after an INSERT, UPDATE, DELETE, SELECT INTO or FETCH statement to determine the number of rows affected by the preceding statement.

Example

```
DECLARE v_rows INTEGER;
...
INSERT INTO mimer_store_book.details ...;
GET DIAGNOSTICS v_rows = ROW_COUNT;
IF v_rows > 0 THEN
```

Note: All SQL statements except GET DIAGNOSTICS will overwrite the information in the diagnostics area.

Using Cursors

You can declare and use cursors in a compound SQL statement to receive a result set from a select-expression or from a result set procedure.

A cursor may not have the same name as another cursor declared in the same scope.

Cursors in a procedural usage context are used in much the same way, in terms of the SQL statements used, as cursors declared outside routines. It is possible to open cursors, fetch data into variables and use the statements UPDATE and DELETE WHERE CURRENT OF cursor.

Example 1

```
DECLARE NREC ROW AS (SOMETABLE);
DECLARE C CURSOR FOR SELECT * FROM SOMETABLE;
BEGIN
  DECLARE EXIT HANDLER FOR NOT FOUND CLOSE C;
  OPEN C;
  LOOP
    FETCH C INTO NREC;
    ...
  END LOOP;
END;
```

Example 2

```
DECLARE D DATE DEFAULT CURRENT_DATE;
DECLARE C1,C2 CHAR(5);
DECLARE Z SCROLL CURSOR FOR CALL PROC(1,D);
DECLARE I INTEGER;
...
OPEN Z;
...
FETCH FIRST FROM Z INTO C1;
...
FETCH ABSOLUTE I FROM Z INTO C2;

FETCH ABSOLUTE I FROM Z INTO C2;
```

Example 1 demonstrates detection of the NOT FOUND exception as a method of detecting that a FETCH statement does not return any data. If a NOT FOUND exception occurs in the example, an exit handler is invoked. After the exit handler has finished, the flow of control leaves the compound SQL statement.

Alternatively, the GET DIAGNOSTICS statement can be used to retrieve the number of rows affected by the FETCH statement, as shown below.

Example

```
DECLARE ROWS INTEGER;
L1:
LOOP
   FETCH X INTO I_CHARGE_CODE,I_AMOUNT;
   GET DIAGNOSTICS ROWCNT = ROW_COUNT;
   IF ROWCNT = 0 THEN
      LEAVE L1;
   END IF;
END LOOP;
CLOSE X;
```

Restrictions

The following specific restrictions apply to cursors used in routines:

- no dynamic functions can be used (i.e. extended cursor names and the use of SQL descriptors)
- REOPENABLE cursors are not allowed
- the use of the keyword RELEASE with the CLOSE statement is not permitted.

Using FETCH to get result set data from a result set procedure may cause parts of the result set procedure to execute, see *Result Set Procedures* on page *264*. The result set procedure will be in use until the associated cursor is closed.

SELECT INTO

Another way of fetching data is by using a SELECT INTO statement. This can only be used when one single row is fetched from the database. If more than one row fulfills the search criteria, an exception condition is raised. If no data is found, a not found condition is raised.

Example

```
SELECT currency, v_price * exchange_rate
   INTO p_local_currency, p_local_price
   FROM mimer_store.customers
       JOIN mimer_store.countries AS cnt ON cnt.code = country_code
           JOIN mimer_store.currencies AS crn ON crn.code = currency_code
   FETCH 1;
```

Transactions

It is possible to start and end transactions within a routine. A transaction is implicitly started when a routine that accesses the database is invoked.

It is also possible to explicitly start a transaction by using the START statement. When a transaction is ended, either by a COMMIT or ROLLBACK statement, all open cursors are closed.

Example

```
START;
UPDATE table
   SET ...
   WHERE col = v_str, ...
...
COMMIT;
```

It is possible to affect the behavior of transactions by using the SET TRANSACTION and SET SESSION statements.

Note: If a compound SQL statement is defined as ATOMIC, a transaction cannot be terminated within it because execution of the COMMIT or ROLLBACK statements is not permitted.

Result Set Procedures

A result set procedure is a special type of procedure that allows a result set to be returned.

A result set procedure is called by specifying it in a cursor declaration and then using FETCH to get the result set data.

In interactive SQL, a result set procedure is called by using the CALL statement and the result set data is dealt with in the same way as a select.

Example (ESQL):

```
EXEC SQL DECLARE c_1 CURSOR FOR CALL result_proc(1, 5);
```

A result set procedure is distinguished when it is created or declared by a RETURNS clause which follows the parameter part of the procedure definition.

The RETURNS clause defines the data types of the columns in the result set and may contain an AS clause which names the columns.

Example

```
CREATE PROCEDURE barcode(IN p_ean BIGINT)
-- result set procedure that returns book or music details for the given EAN
RETURNS TABLE(title VARCHAR(48), creator VARCHAR(48), format VARCHAR(20),
priced decimal(7,2), item_id INTEGER)
READS SQL DATA
BEGIN
   ...
END
```

All result set procedure parameters have mode IN, therefore, any data returned from a result set procedure is returned via the procedure's result set.

The option MODIFIES SQL DATA must not be specified for the access clause of a result set procedure, see *Routine Access Clause* on page *247*.

Note: A function or procedure that has `MODIFIES SQL DATA` specified for its access clause must not be invoked from within a result set procedure.

A result set procedure must not execute a COMMIT or ROLLBACK statement, because this would close the cursor that is used in order to call the result set procedure.

Note: A function or procedure that executes a `COMMIT` or `ROLLBACK` statement must not be invoked from within a result set procedure.

A row in the result set of a result set procedure is returned by executing the `RETURN` statement. The arguments to a `RETURN` statement can be null, an expression or a variable which has the `ROW` data type.

When a `FETCH` is executed, the SQL statements in the body of the result set procedure are executed until a `RETURN` statement is executed.

The execution of the result set procedure is then suspended until the next `FETCH` statement is executed for the calling cursor, then flow of control within the result set procedure continues until the next `RETURN` statement is encountered, or until the end of the procedure is reached.

After flow of control has exited from the scope of a result set procedure the next attempt to `FETCH` more data into the calling cursor will flag end-of-set.

Thus, a result set procedure call can be used in place of the usual `SELECT` when declaring a cursor.

The following example, using ESQL, is intended to demonstrate how execution within the result set procedure proceeds, and is suspended, in response to `FETCH` statements being executed for the calling cursor:

```
EXEC SQL
   CREATE PROCEDURE result_proc(x INTEGER)
   RETURNS TABLE (txt VARCHAR(10), xp INTEGER)
   CONTAINS SQL
   BEGIN
      DECLARE xp INTEGER DEFAULT x;

      RETURN ('FIRST ROW', xp);

      SET xp = x * 2;
      RETURN ('SECOND ROW', xp);

      SET xp = x * 3;
      RETURN ('THIRD ROW', XP);
   END;

EXEC SQL DECLARE c_1 CURSOR FOR CALL result_proc(3);
EXEC SQL OPEN c_1;

EXEC SQL WHENEVER NOT FOUND GOTO done;

EXEC SQL FETCH c_1
            INTO :T, :X;
(This will fetch 'FIRST ROW', 3)
Result set procedure flow of control suspended at XP=X*2

EXEC SQL FETCH c_1
            INTO :T, :X;
(This will fetch 'SECOND ROW', 6)
Result set procedure flow of control suspended at XP=X*3

EXEC SQL FETCH c_1
            INTO :T, :X;
(This will fetch 'THIRD ROW', 9)
Result set procedure flow of control suspended at END;

EXEC SQL FETCH c_1
            INTO :T, :X;
Flow of control exits from procedure scope and the NOT FOUND exception is
raised.

done:
EXEC SQL CLOSE c_1;
```

More typically, a loop construct would be used in the result set procedure to deal with `RETURN` statements. It is also permissible to use a cursor within the result set procedure to get data to be returned via a `SELECT`.

Closing the cursor for a result set procedure will close any open cursors declared within it and no further execution of the procedure will occur.

Reopening the cursor will start execution of the result set procedure afresh from the beginning (i.e. no state information is saved between a close and reopen).

Managing Exception Conditions

An exception is raised if an error occurs when executing an SQL statement. Every exception is identified by an exception condition, expressed in terms of its SQLSTATE value.

About SQLSTATES

An SQLSTATE value is represented by the keyword SQLSTATE followed by a 5-character string containing only uppercase alphanumeric characters. The first two characters of the string identify the exception class and the last three the exception sub-class.

In Mimer SQL, the range of possible SQLSTATE values is divided into standard values and implementation-defined values. The implementation-defined values are those beginning with the characters J-R, T-Z, 5-6 and 8-9. For a list of the values, see *Mimer SQL Reference Manual, Appendix E, SQLSTATE Return Codes*.

Whenever an exception is raised, the exception condition is placed in the diagnostics area and the SQLSTATE value can be retrieved by using the RETURNED_SQLSTATE option of the GET DIAGNOSTICS statement.

Condition Names

In addition to expressing an exception condition in terms of its SQLSTATE value, it is possible (within a compound SQL statement) to declare a condition name to represent it.

Whenever a condition name is used, it is immediately translated into the SQLSTATE value it represents. For more information, see *Declaring Condition Names* on page *268*.

SIGNAL Statements

It is possible to raise an exception without an error occurring by using the SIGNAL statement. When the SIGNAL statement is used, the specified exception condition is placed in the cleared diagnostics area, expressed as its SQLSTATE value, and control proceeds as if an error had just occurred.

It possible to return specific error messages with the SIGNAL statement by using the optional SET clause.

Example

```
SIGNAL SQLSTATE 'UE456'
    SET message_text = 'The specified horse, ' || horse ||
        ' does not exist in the database';
```

Exception Handlers and Actions

It is possible to declare exception handlers in a compound SQL statement that perform some action when exceptions are raised. The action defined by the exception handler is associated with one or more specific exception conditions, or one or more exception class groups, specified when the exception handler is declared. For more information, see *Declaring Exception Handlers* on page *269*.

If there is an exception handler action defined for an exception condition that is raised, the exception handler action is performed and execution continues in the manner defined by the type of the exception handler.

If no exception handler action has been defined for an exception condition that is raised, the default error handling mechanism is invoked (which usually makes the exception condition visible to the calling environment).

If the exception NOT FOUND or an SQLWARNING is raised in an unhandled situation, execution will continue and the exception will be cleared by execution of the next statement in the procedure. The GET DIAGNOSTICS statement can be used to test for the NOT FOUND exception and an SQLWARNING.

RESIGNAL Statements

It may be necessary for an exception handler action to re-raise the current exception condition or to raise an alternative exception condition. The RESIGNAL statement is provided for this purpose and it may only be executed from within an exception handler.

If RESIGNAL is executed without specifying an exception condition, the current exception condition remains in the diagnostics area and the error handling mechanism proceeds to deal with the error as if the current exception handler action had not been found.

If an exception condition is specified (in the same way as for SIGNAL), this is pushed onto the top of the stack of exceptions in the diagnostics area, becoming the current SQLSTATE value, and the error handling mechanism proceeds as just described.

The size of the exceptions stack in the diagnostics area is set by using the SET TRANSACTION DIAGNOSTICS SIZE statement, see *Exception Diagnostics Within Transactions* on page *228*.

Use of RESIGNAL is useful in situations where there are nested exception handler actions defined and it is required that an enclosing exception handler action be invoked from an inner one, or where the default error handling mechanism is to be allowed to proceed from some point within a defined exception handler action. As with the SIGNAL statement it is possible to supply a specific message text.

Example

```
RESIGNAL;

RESIGNAL SQLSTATE 'UE456'
    SET message_text = 'The horse ' || horse || ' does not exist';
```

Declaring Condition Names

As discussed in the previous section, exception conditions are identified by an SQLSTATE value. Whenever an exception is raised, the exception condition that identifies it is stored in the diagnostics area in the form of its SQLSTATE value.

It is always possible to specify an exception condition by using its SQLSTATE value, e.g. SQLSTATE VALUE 'S0700', however it is often desirable to declare a condition name that represents the SQLSTATE value in a way that more meaningfully describes the exception.

Condition names may be declared in a compound SQL statement, see the *Mimer SQL Reference Manual, Chapter 12, COMPOUND STATEMENT*, for a detailed description.

Example

```
DECLARE invalid_parameter CONDITION FOR SQLSTATE 'UE456';
...
SIGNAL invalid_parameter;
```

Following this declaration, the condition name INVALID_PARAMETER can be used instead of the SQLSTATE value SQLSTATE VALUE 'UE456' whenever there is a need to refer to this exception condition.

If a condition name is used in a signal statement the associated SQLSTATE value and the condition name is placed in the diagnostics area. If the condition does not have an associated SQLSTATE value, the SQLSTATE value 45000 is used. A condition is always local to a routine, i.e. consider the following example:

```
create procedure p2()
begin
    declare condition c1;
    ...
    signal c1;
end

create procedure p1()
begin
    declare condition c1;
    declare exit handler for c1
    begin
        ...
        call p2();
        ...
    end
end
```

In this case the exit handler in the procedure p1 will not be invoked when the statement signal c1 is executed.In order to catch a signaled condition the associated SQLSTATE must be used. The condition identifier can be propagated by using a RESIGNAL statement.

All SQLSTATE values in Mimer SQL that lie outside the range of standard values are treated as implementation-defined, so all SQLSTATE values are handled in the same way and may be specified explicitly in all situations.

Declaring Exception Handlers

Exception handlers may be declared in a compound SQL statement in order to define an action which will be executed if specified exceptions are raised within the scope of the exception handler.

The structure of the handler action is the same as the body of a routine, i.e. a single executable procedural SQL statement. The exceptions to which the handler action will respond may be specified as a list of exception conditions or by specifying one or more exception class groups.

The exception class groups are:

- SQLWARNING covers SQLSTATE values beginning with 01.
- NOT FOUND covers SQLSTATE values beginning with 02.
- SQLEXCEPTION covers all other SQLSTATE values (including those in the implementation defined range), excluding those beginning with 00.

An exception handler that is declared to respond to one or more exception class groups is referred to as a general exception handler.

An exception condition may be specified by its `SQLSTATE` value or a condition name declared to represent it. An exception handler which is declared to respond to one or more specific exception conditions is referred to as a specific exception handler.

The same exception condition must not be specified more than once in the same exception handler declaration.

An exception handler can either be a general exception handler or a specific exception handler, i.e. an exception handler declaration cannot contain both exception class groups and specific exception conditions.

Exception handlers are declared in the local handler declaration list of a compound SQL statement and the scope of an exception handler is that compound SQL statement plus all the SQL statements contained within it except when another routine is invoked. When a user defined routine is invoked all exception handlers in the calling routine will get out of scope and they will get into scope again when the invoked routine has finished executing, e.g:

```
CREATE PROCEDURE innerMost(INT x)
BEGIN
    -- no handlers in this routine
    IF  x > 0 THEN
        SIGNAL SQLSTATE 'UE345';
    ELSE
        SIGNAL SQLSTATE 'UE543';
    END IF;
END

CREATE PROCEDURE outerMost()
BEGIN
    DECLARE EXIT HANDLER FOR SQLSTATE 'UE345' BEGIN END;
    DECLARE CONTINUE HANDLER FOR SQLSTATE 'UE543' BEGIN END;
    CALL innerMost(0);
    CALL innerMost(1);
    CALL innerMost(2);
END
```

When the signal statement with exception `UE543` in the `innerMost` routine is executed, the execution of this routine will be stopped as there is no handlers declared. The exception will the be propagated to the `outerMost` routine which has a continue handler for this exception. This means that the execution will proceed with the next call statement. This will cause a new exception (UE345) being signaled. Again this exception will be propagated to the calling routine and the first exception handler will be invoked. As this is an exit handler the execution will continue after the end of the compound statement in the `outerMost` routine, i.e. the statement `call innerMost(2)` will never be executed.

The exception handler will be executed if one of the exceptions it is declared to respond to is raised within the scope of the handler.

A local handler declaration list can only contain one exception handler declared to respond to a particular exception condition or exception class group.

It is possible to declare a general and a specific exception handler, both of which cover the same scope, where an exception condition specified for the specific handler is in one of the exception class groups specified for the general handler. If the exception condition is raised in this situation, the specific handler is executed in preference to the general handler.

It is possible for the scope of two specific exception handlers, which respond to the same exception condition, to overlap. This will be the case if there are two nested compound SQL statements and each declares a specific exception handler for the same exception condition (this is permitted, provided the two exception handlers are not declared in the same local handler declaration list). In this situation the innermost exception handler action will be executed.

The same is true for two general exception handlers in this situation.

The `RESIGNAL` statement can be used in situations like this, in the inner exception handler action, to get the outer exception handler action to execute by propagating the exception out from the exception handler action which is currently executing.

Types of Exception Handlers

Exception handlers fall into the following types:

- **Exit Handler**

  This type of exception handler will execute when the exception condition(s) that apply to it are raised. After the handler has executed, flow of control exits the scope of the compound SQL statement containing the exception handler declaration, by effectively performing a `LEAVE`, see *Scope in Routines – the Compound SQL Statement* on page *248*.

- **Continue Handler**

  This type of exception handler will execute when the exception condition(s) that apply to it are raised. After the handler has executed, flow of control continues by executing the SQL statement immediately following the SQL statement that raised the exception.

- **Undo Handler**

  The execution of this type of handler will be initiated when the exception condition(s) that apply to it are raised. Before the handler action executes, all changes made by the executed SQL statements in the compound SQL statement, or by any SQL statements triggered by them, are canceled. The handler action is then executed and flow of control exits the scope of the compound SQL statement containing the exception handler declaration, by effectively performing a `LEAVE`, see *Scope in Routines – the Compound SQL Statement* on page *248*.

  Note: An `UNDO` exception handler can only be declared in a compound SQL statement that has been defined as `ATOMIC`, see *The ATOMIC Compound SQL Statement* on page *249*.

Examples of Exception Handlers

```
s1:
    BEGIN
        DECLARE EXIT HANDLER FOR SQLEXCEPTION
        BEGIN
            ...
        END;
        ...
    END s1;

s2:
    BEGIN
        DECLARE CONTINUE HANDLER FOR SQLSTATE 'S0700'
        BEGIN
            ...
        END;
        ...
    END s2;

s3:
    BEGIN
        DECLARE EXIT HANDLER FOR SQLSTATE 'S0700'
        BEGIN
            ...
        END;
        ...
s4:
        BEGIN ATOMIC
            DECLARE UNDO HANDLER FOR SQLSTATE 'S0700'
            BEGIN
                ...
            END;
            ...
        END s4;
        ...
    END s3;
```

Create a function iadd that adds two integer values, in an overflow and underflow safe way, using an exit handler. The SQLSTATE value 22003 means “numeric value out of range”:

```
create function iadd(p1 int, p2 int) returns int
begin
    declare exit handler for sqlstate value '22003'
        return case when p1 < 0 and p2 < 0 then -2147483648
                    else 2147483647
                end;
    return p1 + p2;
end
```

Using the GET DIAGNOSTICS Statement

The `GET DIAGNOSTICS` statement can be used in an exception handler to get the specific `SQLSTATE` value that provoked execution of the exception handler.

Example

Create a function iadd that adds two integer values, in an overflow and underflow safe way, using an exit handler. The SQLSTATE value 22003 means "numeric value out of range", the native error -10302 means overflow, and the native error -10303 means underflow:

```
create function iadd(p1 int,p2 int) returns int
begin
    declare exit handler for sqlstate value '22003'
    begin
        declare a int;
        get diagnostics exception 1 a = native_error;
        return case when a = -10303 then -2147483648
                    when a = -10302 then 2147483647
               end;
    end;
    return p1 + p2;
end
```

Example

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
 BEGIN
     DECLARE v_state CHAR(5) DEFAULT '?????';
     GET DIAGNOSTICS EXCEPTION 1 v_state = RETURNED_SQLSTATE;
     CASE v_state
         WHEN '22003' THEN ...
         WHEN '20000' THEN ...
         ELSE RESIGNAL;
     END CASE;
 END;  -- of sqlexception handler
```

Note that `GET DIAGNOSTICS` must be the first statement in the exception handler as the diagnostics area always contains information about the latest statement.

The `GET DIAGNOSTICS` statement can also be used to get information about warnings and not found exceptions.

Example

```
SELECT format into v_format FROM formats where category_id = v_category_id;
GET DIAGNOSTICS EXCEPTION 1 v_state = RETURNED_SQLSTATE;
IF  v_state = '02000' THEN
-- not found
ELSE
-- found
END IF;
```

As mentioned, to describe an error always use GET DIAGNOSTICS in an exception handler. I.e. it is not meaningful to place GET DIAGNOSTICS after a statement to check for errors. Example:

```
BEGIN
    DECLARE v_state CHAR(5);
    DECLARE EXIT HANDLER FOR sqlexception
    BEGIN
        ...
    END;
    INSERT INTO format(format,category_id) VALUES (v_format,v_category_id);
    GET DIAGNOSTICS EXCEPTION 1 v_state = RETURNED_SQLSTATE;
END
```

If an exception occurs in the INSERT statement this will be catched by the exception handler and as this is an exit handler the execution will resume after the compound statement. Thus, the diagnostics statements will never be invoked in this case. Even if it is a continue handler the GET DIAGNOSTICS statement is superfluous as the handler would clear the diagnostics information.

Access Rights

The ident creating a routine must, as is usual, have the appropriate access rights on the tables and other database objects referenced from the SQL statements in the routine. The creating ident must also have the right to create objects in the schema to which the routine is to belong (i.e. the ident must be the creator of the schema).

The right of the creator to access referenced database objects is verified when the CREATE FUNCTION, CREATE MODULE or the CREATE PROCEDURE statement is executed.

If an ident wishes to invoke a routine, that ident must have EXECUTE privilege on the routine.

Note: In order for the creator of a routine to grant EXECUTE privilege on the routine to another ident, the creator must have the WITH GRANT option in affect for all the access rights held on all the database objects referenced within the routine.

The above note is an important security point, because granting EXECUTE privilege on a routine is effectively granting appropriate access rights to the given ident on all the database objects referenced in the routine, therefore all those access rights must be held by the grantor with the WITH GRANT option.

An ident may be granted EXECUTE privilege on a routine with the WITH GRANT option and if this option is in affect, the ident may grant EXECUTE privilege on that routine to other idents.

Routines can be used as a security layer in the database. By having EXECUTE privilege on a routine granted, an ident only gets the right to perform the specific operations specified in the routine and not general access to the referenced database objects.

Note: It is not possible to grant EXECUTE privilege on a module, only on routines.

Using DROP and REVOKE

Care must be taken if database objects are dropped with the `CASCADE` option, and when `REVOKE` is performed, particularly with respect to routines and modules.

It is important to bear in mind the following points in connection with modules and routines:

- Dropping an object referenced by an SQL statement in a routine will cause the routine to be dropped.
- If the access rights on a database object are revoked from the creator of a routine that contains an SQL statement referencing the object, the routine will be dropped.
- If a routine belonging to a module is dropped because of the effects of a cascade, the routine is effectively removed from the module (i.e. the module is not dropped).

If an ident attempts to drop a routine for which there is a compiled version currently being held by another ident, the `DROP` operation will fail because the routine is in use.

When a routine is invoked, it is compiled and the compiled version of the routine is held by the invoking ident. Any other idents invoking a routine while a compiled version of it exists will use the existing compiled version and this will be held by them as well.

A compiled version of a routine will generally be held by an ident until the ident disconnects. If the routine invocation is contained in a dynamic SQL statement, deallocating the statement will release the compiled version of the routine immediately without the need for a disconnect.

The Mimer SQL PSM Debugger

Using Mimer SQL's Java-based graphic PSM Debugger, you can select and run a stored function or stored procedure in an environment that enables you to:

- view the stored source of the routine
- watch the values of the input parameters and declared variables
- observe the results of executing the routine.

You can interactively set breakpoints in the routine being debugged or in other routines or triggers invoked by this routine. When a breakpoint is set the execution will halt at this line, if encountered in the flow of execution.

You can execute a routine in step-wise fashion from the beginning or from the point at which a breakpoint halts execution.

When execution of a routine is interrupted or when a routine is executed in a step-wise fashion, an indicator next to the source line will show the current flow of control position.

Requirements

The PSM Debugger requires a Java 2 (version 1.2 or later) compatible Java runtime environment.

Linux:	The PSM Debugger is a Java-based program and must be used in an X-Windows environment.

Win:	You can download the Java runtime environment files from: https://www.java.com

Starting the PSM Debugger

To start the PSM Debugger:

Linux: Enter the following command:

```
java -jar /usr/lib/psmdebug.jar
```

or, use the `psmdbg` command.

On Linux and macOS, the PSM Debugger can also be started from the **Start Menu**.

VMS: The PSM debugger is not included in the VMS distribution. PSM procedures can be debugged remotely from a workstation using another graphical operating system.

Win: Click **Start**, navigate to where you installed Mimer SQL and select **SQL-PSM Debugger**.

The PSM Debugger window opens and the log in dialog box is displayed.

Logging In

Logging in to the Mimer SQL PSM Debugger establishes a connection to a database server and enables you to access the routines stored in the database.

To log in:

1 In the login dialog box, enter the following details:

Database	The name of a database that is accessible from the node you are running on. The syntax for the database URL in the login dialog box is:
	`hostname[:port]/database`
	If the database resides on your local machine, specify `localhost` as the host name.
	For example:
	`localhost/testdb`

Username	The name of the ident you wish to use to access the database.
Password	The password to be used for the specified ident.

2 Click **OK** to connect to the database, or **Cancel** to quit the Mimer SQL PSM Debugger.

Once you have logged in to the Mimer SQL PSM Debugger, you will be accessing the database as the ident you specified in the log in dialog box and you can choose a routine to debug by selecting it from the Choose a procedure drop-down list.

Choosing a Routine

Choose a stored function or stored procedure to debug by selecting it from the drop-down list.

The list displays the routines for which you hold `EXECUTE` privilege. When you have selected a routine, you must specify values for all the input parameters.

Specifying the Input Parameters

When you select a routine with input parameters, a dialog box will list the name and data type of each input parameter and you must specify values for them.

When you have specified values for all the input parameters, click **OK** to view the source for the routine, or click **Cancel** to go back to selecting a routine from the drop-down list.

Viewing the Source Code for a Routine

When you have selected a routine and specified the input parameter values, the routine source is displayed.

It is possible to watch the values of declared variables and routine parameters declared as input or input/output by selecting them from the Variable details drop-down list.

You can set a breakpoint on a line of the routine source by clicking on the indicator to the left of the source line in the source window.

Watching Variables and Input Parameters

Once you have selected a variable or parameter from the drop-down list, its name and current value (when defined) are shown in the table below the list.

The watched values are updated in the table as the routine executes.

Setting Breakpoints

To set a breakpoint in a routine or trigger invoked by the current routine, you can choose a routine or a trigger from the list in the Breakpoints menu.

When you click on the indicator to left of a source line in the source window, the indicator will change color.

If the indicator is red, there is a breakpoint set on that source line which will halt execution of the routine when it is encountered.

If the indicator next to a source line is white, then no breakpoint is set on that line.

Executing a Routine

To execute a routine:

- Click **Go** to execute the routine to the end or to the next breakpoint.
- Click **Step Into** to execute the next line. If this line contains a call statement or a function invocation, the execution will halt at the first line in the invoked routine.

- Click **Step Over** to execute the next line. This mode will not stop the execution if there is a routine invocation, unless there is a breakpoint set.
- Click **Cancel** to force continuous routine execution to stop.

Whenever the value of a watched variable or parameter changes, the value shown for it in the table will be updated and will be displayed in red. Unchanged values will be displayed in black.

When execution of a routine is halted by a breakpoint, flow of control is positioned just before execution of the source line on which the breakpoint is set. An arrow to the left of the source line shows where execution has been halted.

To continue executing a routine that has been halted at a breakpoint, click **Go**, **Step Into** or **Step Over**.

Whenever execution of a routine is halted, because a breakpoint was encountered or during step-wise execution, the source line at which the routine is halted appears with an arrow to the left of it. The current flow of control position is just prior to execution of that source line.

The results of executing the routine are shown in the window below the routine selection drop-down list. The Mimer SQL PSM Debugger gives the same results feedback during execution of a routine as Mimer BSQL.

Chapter 12

Triggers

This chapter discusses database triggers: how to create them, execute them and drop them.

A trigger defines an SQL statement that is automatically executed before, after, or instead of a specified data manipulation operation on a particular table or view.

A trigger can either be a **statement trigger** which means that the trigger is executed once for a data manipulation statement, or a **row trigger** which means that the trigger is executed once for each row affected by the data manipulation statement causing the trigger.

The execution of the SQL statement can be made conditional on the evaluation of a search condition.

The SQL statement in the trigger definition is typically a compound SQL statement, thus allowing a number of SQL statements to be executed by the trigger. The compound SQL statement must be defined as ATOMIC. Thus, the body of a trigger is similar to the body of a routine and the same language constructs may be used within it. In this code it is possible to refer to the data that was affected by the data manipulation statement which caused the trigger to be executed.

In a statement trigger the affected data is stored in temporary tables. The data in these tables can only be read and not modified. Depending on which event that causes the trigger there can be one or two tables. For delete there is an old table containing all rows that are deleted. For insert there is a new table containing all inserted rows. An update trigger will have both an old and a new table. The old table contains the rows as they were before the update took place while the new table contains the rows as they are after the update has taken place.

In order to be able to refer to these temporary tables, the trigger definition must contain a referencing clause which identifies which names that are used when referencing these tables in any DML statement within the trigger body. The old and new table will have the same layout as the table on which the trigger is defined. An example can be seen below.

A row trigger, depending on the event, will have old row and new row variables that can be referred to in the trigger code. These row variables will have fields with the same name and data type as the columns in the table on which the trigger is defined.

A delete trigger will have an old row variable that contains the deleted row. An insert will have a new row variable that contains the inserted data. An update trigger will have both an old row and a new row variable. Individual data items in these variables are referenced by using dot notation. (See the row trigger example below where `o.country_code` is used to refer to data for the deleted row.) The old row variable is read only but the new row variable can be modified in a before trigger (except that columns defined as large objects are read only in this version of Mimer SQL.)

Creating a Trigger

A trigger is created by using the `CREATE TRIGGER` statement, see the *Mimer SQL Reference Manual, Chapter 12, CREATE TRIGGER*.

Example of statement trigger:

```
CREATE TRIGGER products_after_insert AFTER INSERT ON products
REFERENCING NEW TABLE AS pdt
FOR EACH STATEMENT
BEGIN ATOMIC
   -- Force the update trigger to fire
   UPDATE products
      SET product_search = DEFAULT
      WHERE product_id IN (SELECT product_id
                           FROM pdt);
END -- of trigger products_after_insert
```

A trigger is created on a named table or view and the trigger must be created in the schema to which the table or view belongs.

The trigger name must follow the rules for naming private database objects, see the *Mimer SQL Reference Manual, Chapter 6, Naming Objects*, and the name must be unique within the schema in which the trigger is created.

You can create any number of triggers on a named table, each of which may have the same trigger time, see *Trigger Time* on page *281*, and trigger event, see *Trigger Event* on page *284*, specified.

If two or more triggers exist on the same table with the same trigger time and trigger event, they will be executed in the same order as they were created.

Example of row trigger:

```
create trigger checkExists before delete on currencies
referencing old row as o for each row
    if  exists (select *
                from countries
                where countries.currency_code = o.currency_code) then
        signal sqlstate 'UE123'
            set message_text = 'Depending row in countries exists';
    end if
```

When creating a trigger using the BSQL tool it is convenient to enclose the code as

```
@
create trigger setversion before udpate on document_versions
referencing new row as new_version old row as old_version
begin atomic
    if  old_version.version = new_version.version then
        set new_version.version = new_version.version + 1;
    end if;
end
@
```

thus avoiding conflicts when using `;` as a delimiter in the trigger definition.

Trigger Time

The trigger time specifies when, in relation to the execution of the triggering data manipulation statement, the trigger is executed.

The possible values for the trigger time for a base table are:

- **BEFORE**

  This specifies that the trigger will be executed prior to the execution of the triggering data manipulation statement. The table name must specify a base table which is located in a databank with TRANS or LOG option.

- **AFTER**

  This specifies that the trigger will be executed following the execution of the triggering data manipulation statement.

- **INSTEAD OF**

  For a view it is possible to create instead of triggers. This specifies that the trigger will execute when the triggering data manipulation statement would normally be executed. In this case the triggering data manipulation statement itself has no direct effect, it only causes the trigger to execute.

It is possible to have both row and statement triggers for the same event on a base table. The logic for invoking statement and row triggers for a base table can schematically be seen as:

```
--
-- invoke before statement triggers
--
        call before_statement_trigger_1;
        ...
        call before_statement_trigger_n;

        get_data:
        loop
--
-- get rows affected by data manipulation statement
--
            if  not found then
                leave get_data;
            end if;
--
-- invoke before row triggers
--
            call before_row_trigger_1;
            ...
            call before_row_trigger_n;
--
-- save data to old/new table if used
--

--
-- do actual operation
--
            delete/insert/update;
--
-- invoke after row triggers (currently not supported)
--
            call after_row_trigger_1;
            ...
            call after_row_trigger_n;
        end loop;
```

```
--
-- invoke after statement triggers
--
    call after_statement_trigger_1;
    ...
    call after_statement_trigger_n;
```

Note that this schema includes after row triggers even though these are not supported in this version of Mimer SQL.

Analogously with views, if you have both statement and row trigger the schematical code for invoking triggers would look like

```
    get_data:
    loop
--
-- get rows affected by data manipulation statement
--
        if  not found then
            leave get_data;
        end if;
--
-- execute instead of row trigger
--
        call instead_of_row_trigger_1;
        ...
        call instead_of_row_trigger_n;
--
-- save data to old/new table if used
--
    end loop;
--
-- call instead of statement triggers
--
    call instead_of_statement_trigger_1;
    ...
    call instead_of_statement_trigger_n;
```

Note that this schema includes instead of row triggers even though these are not supported in this version.

Example

Example of an instead of trigger, which can be used for handling join views.

```
CREATE TRIGGER book_details_instead_of_update
   INSTEAD OF UPDATE ON mimer_store_book.book_details
REFERENCING  NEW TABLE AS new_bd
BEGIN ATOMIC
--
-- Update one table with some of the data from the join view
--
   UPDATE titles
      SET authors_list = (SELECT authors_list
                            FROM new_bd
                           WHERE item_id = titles.item_id)
    WHERE item_id IN (SELECT item_id
                        FROM new_bd);
--
-- Update another table using another column from the join view
--
  UPDATE producers
     SET producer_name = (SELECT publisher
                            FROM new_bd
                           WHERE item_id = producers.producer_id)
   WHERE producer_id IN (SELECT item_id
                           FROM new_bd);
END
```

Example

The following example describes how triggers can be used to log all changes made to a table:

```
create table maintab (c1 integer primary key, c2 varchar(10));

create table logtab (ts timestamp default localtimestamp,
                     username nvarchar(128) collate SQL_IDENTIFIER
                         default session_user,
                     operation varchar(6),
                     c1old integer, c2old varchar(10),
                     c1new integer, c2new varchar(10));

@
create trigger maintabinserts after insert on maintab
referencing new table as newt
for each statement
begin atomic
  insert into logtab (operation, c1new, c2new)
    select 'INSERT', newt.c1, newt.c2
    from newt;
end
@

@
create trigger maintabupdates after update on maintab
referencing new table as newt
            old table as oldt
for each statement
begin atomic
  insert into logtab (operation, c1old, c2old, c1new, c2new)
    select 'UPDATE', oldt.c1, oldt.c2, newt.c1, newt.c2
    from oldt, newt
    where oldt.mimer_rowid = newt.mimer_rowid;
end
@
```

```
@
create trigger maintabdeletes after delete on maintab
referencing old table as oldt
for each statement
begin atomic
  insert into logtab (operation, c1old, c2old)
    select 'DELETE', oldt.c1, oldt.c2
    from oldt;
end
@
```

A trigger's old and new tables' rows are sorted in the same order. This means that if old table data and new table data are fetched in parallel, the corresponding rows will be read even if the primary key has been updated.

This example's update trigger uses the `mimer_rowid` pseudo-key to ensure the performance when joining the old and new tables.

Trigger Event

The trigger event specifies the data manipulation statement that will cause the trigger to execute. The possible values for the trigger event are: `INSERT`, `UPDATE` and `DELETE`.

A statement trigger will be executed once each time the specified data manipulation statement is executed on the table on which the trigger was created.

A row trigger will be executed once for each row affected when the specified data manipulation statement is executed on the table on which the trigger was created.

Note: If the trigger time is `INSTEAD OF`, the trigger event itself has no effect on the table (view), it just causes the trigger to execute. The environment executing the trigger event behaves as if the data manipulation statement is actually being executed, even though no changes actually occur in the table(s) that would normally be affected. The only data manipulations possible in this case are those performed by the trigger action.

Trigger Action

The trigger action, like the body of a routine, consists of a single procedural SQL statement. In addition, the execution of the SQL statement can be made conditional on the evaluation of a search condition.

The search condition is specified in the optional `WHEN` clause of the `CREATE TRIGGER` statement.

As for routines, it is recommended that a compound SQL statement always be used for the trigger action.

Note: The entire trigger action must be executed in a single atomic execution context, therefore if a compound SQL statement is used, it must be defined as ATOMIC, see *The ATOMIC Compound SQL Statement* on page *249*.

The SQL statement(s) of the trigger action are always executed within the transaction started for the trigger event. The normal restrictions on the use of certain procedural SQL statements within a transaction apply.

In addition, because the trigger action must be atomic, a `COMMIT` or `ROLLBACK` statement cannot be executed within it.

The creator of the trigger must hold the appropriate access rights, with grant option, for all the operations performed within the trigger action. This is checked when the `CREATE TRIGGER` statement is executed.

If the trigger time specified for the trigger is `BEFORE`, the following restrictions apply to the trigger action:

- the trigger action must not contain any SQL statement that performs data update (i.e. `DELETE`, `INSERT` and `UPDATE` statements are not permitted)
- a routine whose access clause is `MODIFIES SQL DATA` must not be invoked from within the trigger action.

If an exception is raised from the trigger action, it can be handled within the trigger by declaring a handler in the normal way for a compound SQL statement, see *Declaring Exception Handlers* on page *269*.

If there is no handler declared in the trigger action to handle the exception, it will propagate to the environment executing the trigger event and will be dealt with appropriately there. The default behavior at that level will be to undo the effect of the trigger event and all the operations performed in the trigger action.

It is possible to explicitly raise an exception from within the trigger action, or from within an exception handler declared in it, by executing the `SIGNAL` statement.

Altered Table Rows

When the rows of the database table on which the trigger was created are examined from within the trigger action, they will always reflect the actual data manipulations performed by the trigger event and the trigger action.

In the case of an AFTER statement trigger, all rows inserted by the trigger event will be visible, all rows deleted by the trigger event will not be found and all rows updated by the trigger event will appear in their altered state.

In the case of an INSTEAD OF trigger, none of the data manipulations specified by the trigger event will seen when the table is examined because the trigger event does not actually perform any of its data change operations.

The rows of the old table and the new table will always show the changes that were specified by the trigger event, even if these changes were not actually performed on the database table (as is the case for INSTEAD OF triggers).

Recursion

Any data manipulation statements occurring in a trigger action will be executed in the normal way. It is, therefore, possible that the execution of a data manipulation statement in the trigger action may lead to the execution of another trigger or the recursive execution of the current trigger.

In either case, the execution context of the current trigger action is preserved and the newly invoked trigger executes in the normal way, in its own execution context, with appropriate versions of any old table and new table or old row and new row variables.

Example

The following trigger is called recursively. An update statement causes the trigger to fire even when no rows are updated, hence the presence of a when clause to avoid an infinite recursive invocation.

```
CREATE TRIGGER products_after_update
   AFTER UPDATE ON products
REFERENCING NEW TABLE AS pdt
WHEN ( EXISTS (SELECT * FROM pdt) )
BEGIN ATOMIC
   UPDATE products
      SET product_search = product_search_code(product),
          product = (SELECT capitalize(TRIM(product))
                     FROM pdt
                     WHERE product_id = products.product_id)
     WHERE product_id IN (SELECT product_id FROM pdt
                          WHERE product_search <>
                                    product_search_code(products.product)
                            OR product <> capitalize(TRIM(products.product));
END
```

Comments on Triggers

The `COMMENT ON TRIGGER` statement can be used to create a comment on a trigger.

Only the creator of the schema to which the trigger belongs may create a comment on the trigger.

Using DROP and REVOKE

The following points apply to triggers when using `DROP` and `REVOKE` with triggers:

- A trigger can be dropped by using the `DROP TRIGGER` statement.
- Only the creator of the trigger can drop it using the `DROP TRIGGER` statement.
- When a trigger is dropped, the comments created on it are also dropped.
- Dropping an object referenced from an SQL statement in a trigger action will cause the trigger to be dropped.
- If the required privileges held on a database object are revoked from the creator of a trigger whose trigger action contains an SQL statement referencing the object, the trigger will be dropped.

Chapter 13

User-Defined Types And Methods

User-defined types provides a mechanism for defining new data types that can be used in table definitions and stored procedures. For a user-defined type it is possible to create methods belonging to the type. See *Methods* on page *288* for more details.

There are two categories of user-defined types - distinct types and structured types.

Distinct Types

A distinct type is based on a predefined data type.

Example

```
CREATE TYPE size AS INTEGER;
```

This creates a distinct type. When a distinct type is created, there is an implicit creation of a function for converting a value of the type on which the user-defined type is based to the user-defined type. By default this function has the same name as the user-defined type.

Example

```
SELECT name, integer(length) FROM container;
```

It is not possible to compare two instances of different distinct types. This is regardless of if the types on which the distinct types are based are comparable or not. Also, it is not possible to declare an instance of a distinct type with a value of the type on which the distinct type is based. I.e. a statement like

```
SELECT name FROM container WHERE length = 450;
```

is not valid. To do this comparison, either value need to be converted:

```
SELECT name FROM container WHERE integer(length) = 450;
SELECT name FROM container WHERE length = size(450);
```

It is possible to override the default naming of the implicitly created function and method. This is done by using the following syntax:

Example

```
CREATE TYPE size AS integer
    CAST(source as distinct) WITH cast_from_int_to_size
    CAST(distinct as source) WITH cast_from_size_to_int;

INSERT INTO container VALUES ('Large x450', cast_from_int_to_size(450));

SELECT name, cast_from_size_to_int(length) FROM container;
```

Methods

There are three different types of methods - constructor, and instance.

Static methods

A static method does not have the connection with an instance of a user-defined type (like a constructor method has), but works almost the same as a function. The only difference is how they are invoked.

Instance methods

An instance method can only be used with an actual instance of a user-defined type.

Instance methods have an implicit parameter `SELF` that represents the actual value instance used when invoking the method.

Creating Methods

Creating a method is done in two steps, the first is the creation of a method specification and the second is the actual creation.

There can be multiple methods with the same name as long as they either differ by the number of parameters or the type of the parameters.

The method specification can either be given when creating the type or it can be added by using an `ALTER TYPE` statement.

Examples

```
CREATE TYPE bool AS boolean
    CAST(distinct AS source) WITH bool
    METHOD asChar() RETURNS varchar(5);
CREATE METHOD asChar() FOR bool
    RETURN CASE WHEN self THEN 'TRUE' ELSE 'FALSE' END;
```

As can be seen, the type of method is specified in the method specification and the `CREATE METHOD` statement. (The method type can be omitted if an instance method is created.) It is possible to use all PSM statements when creating a method. The for clause with the type name is needed since it is possible to have methods with the same name and parameters for different user-defined types.

There are some specific rules for constructor methods, the method name must be the same as the name of the user-defined type and the return type must be the user-defined type.

A method specification can be dropped from a type by using a variant of the alter type statement. If there are any method defined using that method specification, that method will be dropped if cascade is specified.

Example

```
ALTER TYPE ymd DROP CONSTRUCTOR METHOD ymd(date) CASCADE;
ALTER TYPE ymd DROP STATIC METHOD add RESTRICT;
```

If neither `RESTRICT` nor `CASCADE` is specified, `RESTRICT` is default. Note that it is necessary to specify the data type of the parameters if there are multiple methods with the same name and method type for the user-defined type. The second example will only be successful if there is only one static method named add for the type ymd. To specify a method with no parameters an empty pair of parenthesis () can be used.

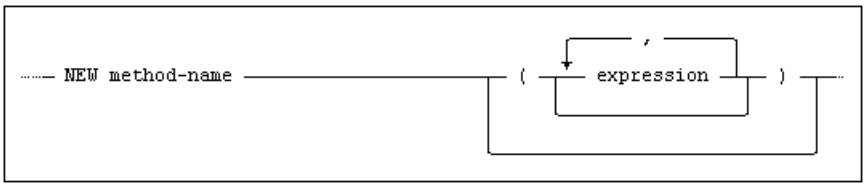
See *Mimer SQL Reference Manual, Chapter 12, CREATE METHOD* for additional information.

Invoking Methods

Methods are invoked differently depending on the type of the method.

Invoking a constructor method

A constructor method is used with the `NEW` operator when creating a new instance of a user-defined type.



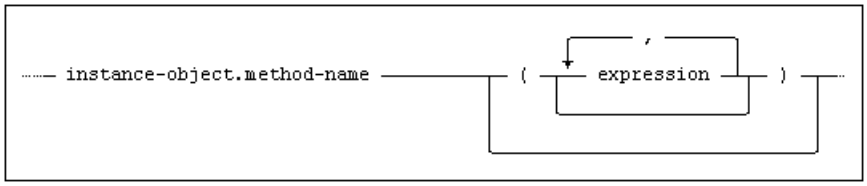
When using the `NEW` operator, the constructor function will be invoked first. This function assigns default values to all attributes. The constructor function returns the instance and this will be passed as an implicit parameter to the constructor method together with the explicit parameter. The constructor method modifies the attributes and returns the instance. It is possible to have a constructor method without parameters.

Example

```
BEGIN
    DECLARE a ymd;

    SET a = NEW ymd(CURRENT_DATE);
END
```

Invoking an instance method

An instance method is invoked by using dot notation on a expression that evaluates to a user-defined type.

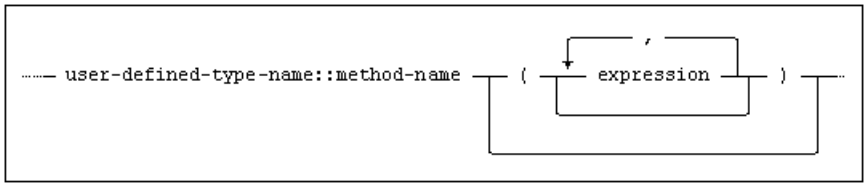


Example

```
BEGIN
    DECLARE a ymd;
    DECLARE b int;

    SET a = NEW ymd(DATE'2010-04-23');
--
-- invoke the implicitly created instance method m
-- for retrieving the value of the attribute m
--
    SET b = a.m()

    ...
--
-- as the instance method compare returns a user-defined type,
-- the method asChar for this user-defined type can be invoked on that result
--
    IF a.compare(2010,11,2).asChar() = 'TRUE' THEN
    ...
    END IF;
END
```

Invoking a static method

A static method is invoked with a double-colon syntax.



Example

```
SELECT ymd::add(c) FROM t;
```

Dropping Methods

Dropping a method will have effects on objects using it. The `DROP` statement may either have a restrict or cascade option. Restrict means that if there are any objects depending on the method the drop will not be done. If cascade is specified all such objects will be dropped.

Example

```
DROP STATIC METHOD add FOR ymd CASCADE;
```

See *Mimer SQL Reference Manual, Chapter 12, DROP* for details.

Chapter 14

Spatial Data

The pre-defined schema `BUILTIN` contains user-defined types and methods used to store and search spatial data in an efficient manner. The functionality allows positions to be indexed and searched quickly.

There are two basic groups of spatial data:

- Geographical data, i.e. positions on the surface of the Earth. See *Geographical Data* on page *291*.
- Coordinate system data, i.e. positions in a two-dimensional plane. See *Coordinate System Data* on page *301*.

Geographical Data

The following user-defined types are used to store geographical data:

Type	SQL type	Description
`BUILTIN.GIS_LATITUDE`	`BINARY(4)`	A distinct user-defined type that stores latitude values. See *BUILTIN.GIS_LATITUDE* on page *291*.
`BUILTIN.GIS_LONGITUDE`	`BINARY(4)`	A distinct user-defined type that stores longitude values. See *BUILTIN.GIS_LONGITUDE* on page *294*.
`BUILTIN.GIS_LOCATION`	`BINARY(8)`	A distinct user-defined type that is used to store a location on Earth. It has a latitude and a longitude component. See *BUILTIN.GIS_LOCATION* on page *297*.

BUILTIN.GIS_LATITUDE

The `builtin.gis_latitude` data type is used to store latitude values. Valid values are between -90° and 90°. Negative values denote south latitudes, and positive values north latitudes.

The following routines belong to the user-defined type:

Routine	Routine type	Description
BUILTIN.GIS_LATITUDE	Function (constructor)	Creates an instance of the type, with decimal input.
AS_DECIMAL	Instance method	Returns data as a decimal value.
AS_DOUBLE	Instance method	Returns data as a double precision value.
AS_TEXT	Instance method	Returns data as text, DDMMSS.ssss format, with a leading N for north or S for south.
AS_TEXT(fmt)	Instance method	Returns data as text, on a format specified by the fmt parameter: 1 = DDMMSS.ssss format, with a leading + for north and – for south 2 = DDMMSS.ssss format, with a leading N for north and S for south 3 = DD°MM'SS.ssss" format, with a trailing N for north and S for south.

Example

Create a table and insert a few values

```
create table latitudes (lat builtin.gis_latitude, description varchar(40));
```

Use builtin.gis_latitude to convert input values.

```
insert into latitudes values (builtin.gis_latitude(0),'Equator');
insert into latitudes values (builtin.gis_latitude(66.5619),'Arctic Circle');
insert into latitudes values (builtin.gis_latitude(-23.4389),
                              'Tropic of Capricorn');
```

Add an index to ensure search performance.

```
create index latx on latitudes (lat);
```

Read the latitude values, without any conversion

```
SQL>select * from latitudes;
lat      description
======== ========================================
80000000 Equator
A7AC8A38 Arctic Circle
720781F8 Tropic of Capricorn

                    3 rows found
```

Read the latitude values as decimal

Use the `as_decimal` method to return the data as a decimal value.

```
SQL>select lat.as_decimal(), description from latitudes;
            description
=========== ======================================
  0.0000000 Equator
 66.5619000 Arctic Circle
-23.4389000 Tropic of Capricorn

                  3 rows found
```

Read the latitude values as double precision

Use the `as_double` method to return the data as a double precision value.

```
SQL>select lat.as_double(), description from latitudes;
                         description
======================== ======================================
 0.0000000000000000E+000 Equator
 6.6561899999999994E+001 Arctic Circle
-2.3438900000000000E+001 Tropic of Capricorn

                  3 rows found
```

Return the latitude values as character, default format

Use the `as_text` method to return the data as DDMMSS.ssss text, with a leading `N` for north or `S` for south.

```
SQL>select lat.as_text(), description from latitudes;
                    description
=================== ======================================
N000000.0000        Equator
N663339.9000        Arctic Circle
S232616.9000        Tropic of Capricorn

                  3 rows found
```

Return the latitude values as character, N/S format

The `as_text` method with input value 1 will return data as DDMMSS.ssss text, with `N` for north and `S` for south.

```
SQL>select lat.as_text(1), description from latitudes;
                    description
=================== ======================================
N000000.0000        Equator
N663339.9000        Arctic Circle
S232616.9000        Tropic of Capricorn

                  3 rows found
```

Return the latitude values as character, +/- format

The `as_text` method with input value 2 will return the data as DDMMSS.ssss text, with + for north and – for south.

```
SQL>select lat.as_text(2), description from latitudes;
                    description
=================== ======================================
+000000.0000        Equator
+663339.9000        Arctic Circle
-232616.9000        Tropic of Capricorn

                  3 rows found
```

Return the latitude values as character, traditional format

The `as_text` method with input value 3 will return the data as character, with ° for degrees, ' for minutes and '' for seconds.

```
SQL>select lat.as_text(3), description from latitudes;
                     description
==================== ========================================
00°00'00.0000''N     Equator
66°33'39.9000''N     Arctic Circle
23°26'16.9000''S     Tropic of Capricorn

                  3 rows found
```

SELECT the latitude values north of latitude N60

Use `builtin.gis_latitude` for input values.

```
SQL>select lat.as_decimal(), description
SQL&from latitudes
SQL&where lat > builtin.gis_latitude(60);
            description
=========== ========================================
 66.5619000 Arctic Circle

                  1 row found
```

BUILTIN.GIS_LONGITUDE

The `builtin.gis_longitude` data type is used to store longitude values. Valid values are between -180° and 180°. Negative values denote west longitudes, and positive values east longitudes.

The following routines belong to the user-defined type:

Routine	Routine type	Description
`BUILTIN.GIS_LONGITUDE`	Function (constructor)	Creates an instance of the type, with decimal input.
`AS_DECIMAL`	Instance method	Returns data as a decimal value.
`AS_DOUBLE`	Instance method	Returns data as a double precision value.
`AS_TEXT`	Instance method	Returns data as text, DDDMMSS.ssss format, with a leading `E` for east and `W` for west.

Routine	Routine type	Description
`AS_TEXT(fmt)`	Instance method	Returns data as text, on a format specified by the `fmt` parameter: `1` = DDDMMSS.ssss format, with a leading + for east and – for west `2` = DDDMMSS.ssss format, with a leading `E` for east and `W` for west `3` = DDD°MM'SS.ssss" format, with a trailing `E` for east and `W` for west.

Examples

Create a table and insert a few values

```
create table longitudes (long builtin.gis_longitude, description varchar(40));
```

Use `builtin.gis_longitude` to convert input values.

```
insert into longitudes values (builtin.gis_longitude(0),'Prime Meridian');
insert into longitudes values (builtin.gis_longitude(-110.0),'Saskatchewan, W');
insert into longitudes values (builtin.gis_longitude(141.0),'South Australia, E');
```

Add an index to ensure search performance.

```
create index longx on longitudes (long);
```

Read the longitude values, without any conversion

```
SQL>select * from longitudes;
long     description
======== ========================================
80000000 Prime Meridian
3E6F5500 Saskatchewan, W
D40AE480 South Australia, E

                  3 rows found
```

Read the longitude values as decimal

Use the `as_decimal` method to return the data as decimal.

```
SQL>select long.as_decimal(), description from longitudes;
             description
============ ========================================
   0.0000000 Prime Meridian
-110.0000000 Saskatchewan, W
 141.0000000 South Australia, E

                  3 rows found
```

Read the longitude values as decimal

Use the `as_double` method to return the data as double precision.

```
SQL>select long.as_decimal(), description from longitudes;
            description
============ ======================================
   0.0000000 Prime Meridian
-110.0000000 Saskatchewan, W
 141.0000000 South Australia, E

                3 rows found
```

Return the longitude values as character, default format

The `as_text` method with input value 1 will return data as DDDMMSS.ssss text, with a leading `E` for east and `W` for west.

```
SQL>select long.as_text(), description from longitudes;
                    description
=================== ======================================
E0000000.0000       Prime Meridian
W1100000.0000       Saskatchewan, W
E1410000.0000       South Australia, E

                3 rows found
```

Return the longitude values as character, E/W format

Use the `as_text` method to return the data as DDDMMSS.ssss text, with a leading `E` for east and `W` for west.

```
SQL>select long.as_text(1), description from longitudes;
                    description
=================== ======================================
E0000000.0000       Prime Meridian
W1100000.0000       Saskatchewan, W
E1410000.0000       South Australia, E

                3 rows found
```

Return the longitude values as character, +/- format

The `as_text` method with input value 2 will return the data as DDDMMSS.ssss text, with a leading + for east and – for west.

```
SQL>select long.as_text(2), description from longitudes;
                    description
=================== ======================================
+0000000.0000       Prime Meridian
-1100000.0000       Saskatchewan, W
+1410000.0000       South Australia, E

                3 rows found
```

Return the longitude values as character, traditional format

The `as_text` method with input value 3 will return the data as DDD°MM'SS.ssss" text, with a trailing `E` for east and `W` for west.

```
SQL>select long.as_text(3), description from longitudes;
                     description
=================== ========================================
000°00'00.0000''E    Prime Meridian
110°00'00.0000''W    Saskatchewan, W
141°00'00.0000''E    South Australia, E

                 3 rows found
```

SELECT the longitude values between longitude W60 and W30

Use `builtin.gis_longitude` for input values.

```
SQL>select long.as_decimal(), description
SQL&from longitudes
SQL&where long between builtin.gis_longitude(-120)
SQL&               and builtin.gis_longitude(-90);
             description
============ ======================================
-110.0000000 Saskatchewan, W

                 1 row found
```

BUILTIN.GIS_LOCATION

The distinct user-defined type `builtin.gis_location` is used to store a location on Earth. It has a latitude component and a longitude component. (See *BUILTIN.GIS_LATITUDE* on page *291* and *BUILTIN.GIS_LONGITUDE* on page *294* for details.)

The following routines belong to the user-defined type:

Routine	Routine type	Description
`BUILTIN.GIS_LOCATION(lat,long)`	Function (constructor)	Creates an instance of the type. The `lat` parameter is for latitude, the `long` parameter is for longitude.
`AS_TEXT`	Instance method	Returns "latitude, longitude" data as DDMMSS.ssss values text, with `N`/`S` and `E`/`W` notation.

Routine	Routine type	Description
`AS_TEXT(fmt)`	Instance method	Returns “latitude, longitude” data on a format specified by the `fmt` parameter: `1` = DDMMSS.ssss formats, with `+` for east and north, and `-` for west and south `2` = DDMMSS.ssss formats, with leading `N` for north and `S` for south, and `E` for east and `W` for west `3` = DD°MM'SS.ssss" formats, with trailing `N` for north and `S` for south, and trailing `E` for east and `W` for west
`LATITUDE`	Instance method	Used to retrieve the latitude part of the location.
`LONGITUDE`	Instance method	Used to retrieve the longitude part of the location.
`INSIDE_RECTANGLE(ll,ur)`	Instance method	Method that returns whether a location is inside a rectangular area of the map. The `ll` parameter is for the lower left corner of the rectangle, and the `ur` parameter is for the upper right corner. May use indexes when available.

Example

Create a table and insert a few values

```
create table locations (location builtin.gis_location, place nvarchar(30));
```

Use `builtin.gis_location` for input values.

```
insert into locations values
  (builtin.gis_location(40.752134,-73.974638),'Chrysler Building');
insert into locations values
  (builtin.gis_location(40.6892,-74.0445),'Statue of Liberty');
insert into locations values
  (builtin.gis_location(40.735681,-73.99043),'Union Square');
insert into locations values
  (builtin.gis_location(40.829167,-73.926389),'Yankee Stadium');
insert into locations values
  (builtin.gis_location(40.756,-73.987),'Times Square');
insert into locations values
  (builtin.gis_location(40.767778,-73.971667),'Central Park Zoo');
insert into locations values
  (builtin.gis_location(40.729861,-73.991434),'Astor Place');
insert into locations values
  (builtin.gis_location(40.779447,-73.96311),'Metropolitan Museum');
insert into locations values
  (builtin.gis_location(40.782975,-73.958992),'Guggenheim Museum');
insert into locations values
  (builtin.gis_location(40.703717,-74.016094),'Battery Park');
```

Add an index to ensure search performance.

```
create index locx on locations (location);
```

Read the inserted data, as it is

Use no conversion, just read raw data.

```
SQL>select * from locations;
location         place
================ ==============================
938574C831D14FB0 Chrysler Building
93857151CDB2ED40 Statue of Liberty
9385743BF532D198 Union Square
9385767D45C6367C Yankee Stadium
9385749CB7EE7100 Times Square
938574E0D98B7224 Central Park Zoo
93857439F8594B58 Astor Place
938574EC2E0A0838 Metropolitan Museum
938574ECFAA0FE28 Guggenheim Museum
9385740CA864BB18 Battery Park

                 10 rows found
```

Read the inserted data, as text

Use the `as_text` method to return more readable locations.

```
SQL>select location.as_text(), place from locations;
                                          place
======================================== ==============================
N404507.1340,W0735826.6380               Chrysler Building
N404120.2000,W0740238.5000               Statue of Liberty
N404406.6810,W0735924.4300               Union Square
N404944.1670,W0735533.3890               Yankee Stadium
N404521.0000,W0735913.0000               Times Square
N404601.7780,W0735815.6670               Central Park Zoo
N404344.8610,W0735927.4340               Astor Place
N404644.4470,W0735746.1100               Metropolitan Museum
N404655.9750,W0735728.9920               Guggenheim Museum
N404210.7170,W0740057.0940               Battery Park

                10 rows found
```

Read the inserted data, as decimal

Use the `as_decimal` methods to return the locations' latitude and longitude components.

```
SQL>select location.latitude().as_decimal(),
SQL>       location.longitude().as_decimal(),
SQL>       place
SQL&from locations;
                        place
========== ============ ==============================
 40.7521340  -73.9746380 Chrysler Building
 40.6892000  -74.0445000 Statue of Liberty
 40.7356810  -73.9904300 Union Square
 40.8291670  -73.9263890 Yankee Stadium
 40.7560000  -73.9870000 Times Square
 40.7677780  -73.9716670 Central Park Zoo
 40.7298610  -73.9914340 Astor Place
 40.7794470  -73.9631100 Metropolitan Museum
 40.7829750  -73.9589920 Guggenheim Museum
 40.7037170  -74.0160940 Battery Park

                10 rows found
```

Find the locations inside an area

Use the `inside_rectangle` method to find locations. Remember that `builtin.gis_location` wants decimal input!

```
SQL>select location.as_text(1), place from locations
SQL&where location.inside_rectangle(builtin.gis_location(40.75,-74.0),
SQL&                                builtin.gis_location(40.80,-73.0));
                                          place
======================================== ==============================
N404507.1340,W0735826.6380               Chrysler Building
N404521.0000,W0735913.0000               Times Square
N404601.7780,W0735815.6670               Central Park Zoo
N404644.4470,W0735746.1100               Metropolitan Museum
N404655.9750,W0735728.9920               Guggenheim Museum

                 5 rows found
```

Find the same locations, but order them from south to north.

```
SQL>select location.as_text(1), place from locations
SQL&where location.inside_rectangle(builtin.gis_location(40.75,-74.0),
SQL&                                builtin.gis_location(40.80,-73.0))
SQL&order by location.latitude;
                                          place
======================================== ==============================
N404507.1340,W0735826.6380               Chrysler Building
N404521.0000,W0735913.0000               Times Square
N404601.7780,W0735815.6670               Central Park Zoo
N404644.4470,W0735746.1100               Metropolitan Museum
N404655.9750,W0735728.9920               Guggenheim Museum

                  5 rows found
```

Coordinate System Data

The following user-defined type is used to store coordinate system data:

Type	SQL type	Description
BUILTIN.GIS_COORDINATE	BINARY(8)	This type has an x and a y component in a flat coordinate system. See *BUILTIN.GIS_COORDINATE* on page *301*.

BUILTIN.GIS_COORDINATE

The distinct user-defined type `builtin.gis_coordinate` is used to store points in a two dimensional coordinate system. This type has an x and a y component, both represented by an integer value.

The following routines belong to the user-defined type:

Routine	Routine type	Description
BUILTIN.GIS_COORDINATE(x,y)	Function (constructor)	Creates an instance of the type.
X	Instance method	Used to retrieve the x unit of the point.
Y	Instance method	Used to retrieve the y unit of the point.
INSIDE_RECTANGLE(ll,ur)	Instance method	Method that returns whether a point is inside a rectangular area of the coordinate system. The `ll` parameter is for the lower left corner of the rectangle, and the `ur` parameter is for the upper right corner. May use indexes when available.

Example

Create a table and insert a few values

```
create table coordinates (id integer primary key, point builtin.gis_coordinate);
```

Use `builtin.gis_coordinate` to insert values.

```
insert into coordinates values (1, builtin.gis_coordinate(25,15));
insert into coordinates values (2, builtin.gis_coordinate(30,40));
insert into coordinates values (3, builtin.gis_coordinate(-3,33));
insert into coordinates values (4, builtin.gis_coordinate(-40,-55));
insert into coordinates values (5, builtin.gis_coordinate(115,25));
insert into coordinates values (6, builtin.gis_coordinate(5,125));
insert into coordinates values (7, builtin.gis_coordinate(-5,125));
insert into coordinates values (8, builtin.gis_coordinate(0,25));
insert into coordinates values (9, builtin.gis_coordinate(76,-1));
insert into coordinates values (10, builtin.gis_coordinate(100,100));
```

Add an index to ensure search performance.

```
create index coordsx on coordinates (point);
```

Read the inserted data, as it is

Use no conversion, just read raw data.

```
SQL>select * from coordinates;
         id point
========== ================
          1 C0000000000001EB
          2 C0000000000009D4
          3 9555555555555D53
          4 3FFFFFFFFFFF1C2
          5 C000000000001787
          6 C000000000002AB3
          7 9555555555557FE7
          8 C000000000000282
          9 6AAAAAAAAAABAFA
         10 C000000000003C30

                10 rows found
```

Read the x and y values

Use the `x` and `y` methods to return more readable points.

```
SQL>select id, point.x() as x, point.y() as y from coordinates;
         id           x           y
========== =========== ===========
          1          25          15
          2          30          40
          3          -3          33
          4         -40         -55
          5         115          25
          6           5         125
          7          -5         125
          8           0          25
          9          76          -1
         10         100         100

                10 rows found
```

Find the points inside an rectangle

Use the `inside_rectangle` method to find points inside an rectangle.

```
SQL>select id, point.x() as x, point.y() as y from coordinates
SQL&where point.inside_rectangle(builtin.gis_coordinate(0,0),
SQL&                             builtin.gis_coordinate(100,100));
         id           x           y
=========== =========== ===========
          1          25          15
          8           0          25
          2          30          40
         10         100         100

                  4 rows found
```

Find the same points, but order them by the `x` coordinate.

```
SQL>select id, point.x() as x, point.y() as y from coordinates
SQL&where point.inside_rectangle(builtin.gis_coordinate(0,0),
SQL&                             builtin.gis_coordinate(100,100))
SQL&order by point.x;
         id           x           y
=========== =========== ===========
          8           0          25
          1          25          15
          2          30          40
         10         100         100

                  4 rows found
```

Chapter 15

Universally Unique Identifier - UUID

The pre-defined schema `BUILTIN` contains the following user-defined type to store unique identifier data:

Type	SQL type	Description
`builtin.uuid`	`BINARY(16)`	

The following routines belong to the uuid user-defined type:

Routine	Routine type	Description
`BUILTIN.UUID_NEW()`	Function	Generates a new uuid value, which with very high probability is globally unique.
`BUILTIN.UUID_FROM_TEXT(HEXSTRING)`	Function	Converts a hexadecimal string to an uuid value. The HEXSTRING argument should follow the standard format `hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh`, where each `h` is a hexadecimal value represented by using the characters 0-9 and a-f.
`AS_TEXT`	Instance method	Converts a uuid value to a character representation, using the UUID standard format.

Example

Create a table and insert a few values:

```
create table tuuid (uuid builtin.uuid);

insert into tuuid values (builtin.uuid_new());
insert into tuuid values (builtin.uuid_new());

select uuid.as_text() from tuuid;

====================================
039f01f0-a635-11e8-9f4c-aa0004007a04
04494bb0-a635-11e8-9f4c-aa0004007a04
```

It is also possible to insert uuid values explicitly, either by using a binary constant or converting a character string using the function `bultin.uuid_from_text`.

```
insert into tuuid values (x'62E4BD10A63711E8A174AA0004007A04');
insert into tuuid values
    (builtin.uuid_from_text('85dc2790-a637-11e8-a174-aa0004007a04'));
```

Create an index to improve search performance:

```
create index tuuid_ix on tuuid (uuid);

select * from tuuid where uuid =
    builtin.uuid_from_text('85dc2790-a637-11e8-a174-aa0004007a04'));
```

More information about uuid can be found at
https://en.wikipedia.org/wiki/Universally_unique_identifier

Appendix A

Host Language Dependent Aspects

You can use embedded SQL (ESQL) statements in any of the following host languages:

- C/C++
- COBOL
- Fortran

Note: It is not a complete description of the rules for writing ESQL programs. The programmer should use the main body of this manual as a guide to writing programs, and refer to this appendix for language-specific details.

The following topics are discussed for each language:

- SQL statement format: delimiters, margins, line continuation, comments, special characters.
- Restrictions.
- Host variables - declarations, SQL data type correspondence, value assignment rules.
- Preprocessor output format.
- Scope rules.

This appendix describes features of ESQL that differ between the respective host languages.

ESQL in C/C++ Programs

Mimer SQL supports ESQL for C/C++ following the ISO/ANSI standard.

SQL Statement Format

The following sections discuss the SQL statement format.

Statement Delimiters

SQL statements are identified by the leading delimiter `EXEC SQL` and terminated by a semicolon `;`, for example:

```
EXEC SQL DELETE FROM countries;
```

Line Continuation

Line continuation rules for SQL statements are the same as those for ordinary C statements.

For a string constant, a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. `<TAB>, <LF>, <VT>, <FF>, <CR> or <SP>`), can be used to join two or more sub-strings. Each substring must be separately enclosed in delimiters. For example:

```
EXEC SQL COMMENT ON TABLE currencies IS 'Holds currency'<CR>
                                        ' details';
```

Comments

Comments, from `//` to end-of-line, or enclosed between the markers `/*` and `*/`, may be written anywhere within SQL statements where a white-space is permitted, except between the keywords `EXEC` and `SQL` and within string constants. The comment may replace the white-space, for example:

```
EXEC SQL DELETE/* all rows */FROM countries;
```

Special Characters

The delimiters in SQL are single quotation marks `'` for string constants and double quotation marks `"` for delimited identifiers. This is contrary to the C string delimiter usage.

```
EXEC SQL INSERT INTO "tablename" VALUES ('text string');
```

A white-space character separates keywords.

Host Variables in C/C++

The following sections discuss declarations, SQL data type correspondence and value assignments.

Declarations

Host variables used in SQL statements must be declared within the `SQL DECLARE SECTION`, delimited by the statements `BEGIN DECLARE SECTION` and `END DECLARE SECTION`.

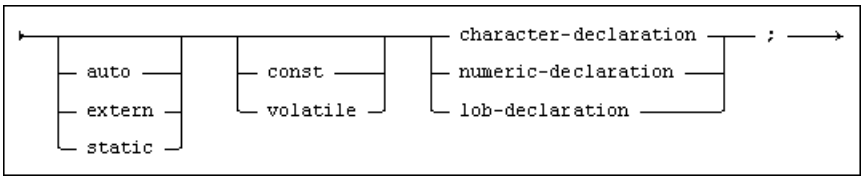
Variables declared within the `SQL DECLARE SECTION` must conform to the following rules in order to be recognized by the SQL preprocessor:

- host variables may be of `AUTO`, `EXTERN` or `STATIC` class
- array variables are not permitted with the exception of character arrays
- character arrays are interpreted as null terminated strings. The hostvariable should be declared with a length one greater than the length of the column, because of the null termination
- the `VARCHAR` host variable data type is recognized by the ESQL/C preprocessor and should be used when variable-length character data is to be returned from SQL as a null terminated string without any blank padding (the `VARCHAR` host variable should be declared with a length one greater than the length of the column, because of the null termination).
- the `NCHAR` host variable data type is recognized by the ESQL/C preprocessor and should be used when Unicode data is to be returned from SQL as a null terminated string with blank padding. The `NCHAR` host variable should be declared with a length one greater than the length of the column, because of the null termination.
- the `NCHAR VARYING` host variable data type is recognized by the ESQL/C preprocessor and should be used when variable-length Unicode data is to be returned from SQL as a null terminated string without any blank padding. The `NCHAR VARYING` host variable should be declared with a length one greater than the length of the column, because of the null termination.
- where binary data is stored in a character array, the size of the array must match the length of the binary data exactly because binary data is not terminated and therefore all array elements are significant
- variable names are case significant
- indicator variables should be declared as `short` or `int`
- `SQLSTATE` should be declared as `char[6]` or `VARCHAR[6]`
- Only data types `CHAR`, `VARCHAR`, `NCHAR`, `NCHAR VARYING`, `BLOB`, `CLOB` and `NCLOB` can be indexed.

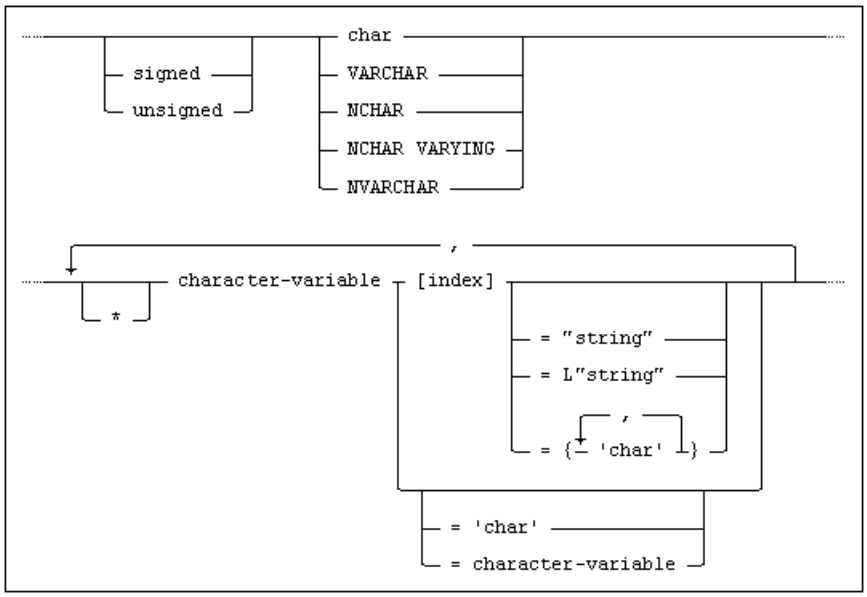
When reading any character array host variable, declared as `CHAR`, `VARCHAR`, `NCHAR` or `NCHAR VARYING`, the contents of the variable must be null terminated. When a host variable declared as `CHAR` or `NCHAR` is read, its value is blank padded to the same length as the host variable. When a host variable declared as `VARCHAR` or `NCHAR VARYING` is read, no blank padding is performed.

When any type conversion is done when retrieving a numeric value to a fixed length character host variable, i.e. `CHAR` or `NCHAR`, the data will be right justified. When type conversion is done when retrieving a value to a variable length character type host variable, i.e. `VARCHAR` or `NCHAR VARYING`, the data will be left justified.

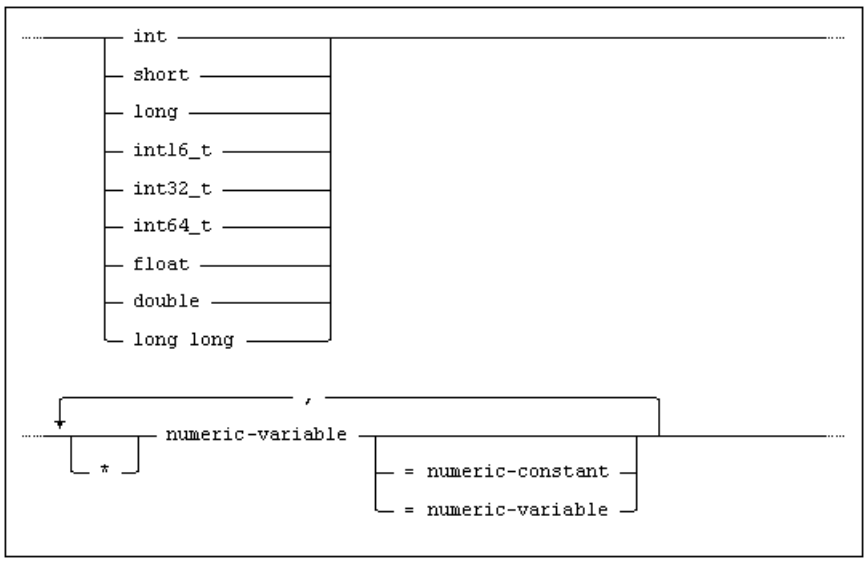
A syntax diagram showing the variable declarations recognized by the ESQL/C preprocessor is given below:



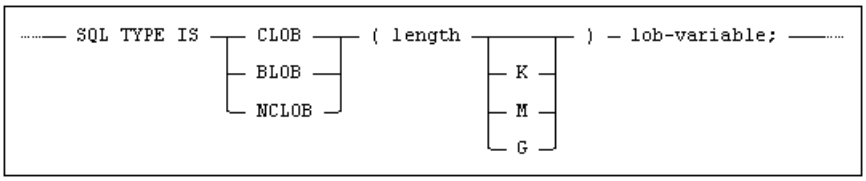
where `character-declaration` is:



and `numeric-declaration` is:



and `lob-declaration` is:



The following points should be noted:

- In accordance with the syntax rules of C, keywords are case-sensitive and are given in the required case in the syntax diagram. This deviates from the general practice in Mimer SQL documentation of using upper-case to denote keywords
- `Index` must be a number which is 1 or greater

SQL Data Type Correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

SQL data type	C variable declaration
SMALLINT INTEGER BIGINT	short int16_t int int32_t long long long int64_t
DECIMAL	float double
FLOAT REAL DOUBLE PRECISION	float double
CHARACTER VARCHAR DATETIME INTERVAL	char varchar [a]
NCHAR NCHAR VARYING	nchar [b] wchar_t nchar varying[c] varwchar_t[d]
BINARY BINARY VARYING BLOB	sql type is blob [e]
CLOB	sql type is clob [f]
NCLOB	sql type is nclob [g]

a. The `varchar` host variable type is recognized by the ESQL/C preprocessor and converted to the `char` data type in C.
b. The `nchar` host variable type is recognized by the ESQL/C preprocessor and converted to the `wchar_t` data type in C.
c. The `nchar varying` host variable type is recognized by the ESQL/C preprocessor and converted to the `wchar_t` data type in C.
d. The `varwchar_t` host variable type is recognized by the ESQL/C preprocessor and converted to the `wchar_t` data type in C.

e. The `blob` host variable type is recognized by the ESQL/C preprocessor and converted to:

```
struct {
   long            hvn_reserved;
   unsigned long   hvn_length;
   char            hvn_data[L];
   } hvn
```

Where *L* is the numeric value of the large object length and *hvn* is the host variable name as specified in the `lob-declaration`.

f. The `clob` host variable type is recognized by the ESQL/C preprocessor and converted to:

```
struct {
   long            hvn_reserved;
   unsigned long   hvn_length;
   char            hvn_data[L];
   } hvn
```

Where *L* is the numeric value of the large object length and *hvn* is the host variable name as specified in the `lob-declaration`.

g. The `nclob` host variable type is recognized by the ESQL/C preprocessor and converted to:

```
struct {
   long            hvn_reserved;
   unsigned long   hvn_length;
   wchar_t         hvn_data[L];
   } hvn
```

Where *L* is the numeric value of the large object length and *hvn* is the host variable name as specified in the `nclob-declaration`

Note: Your C compiler may not support all of these possible declarations.

IEEE floating point

Mimer supports single precision and double precision IEEE floating point values stored in the C types float and double. Values are stored exactly in the database with the following exceptions:

- -0.0 is stored as 0.0
- NaN (Not a Number) or Inf (Infinity) values are not permitted in the database and will not be stored.

Value Assignments

The general rules for conversion of values between compatible but different data types, see the *Mimer SQL Reference Manual, Chapter 6, SQL Syntax Elements*, apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

When reading any character array host variable, the contents of the variable must be null terminated. When a fixed length character host variable (`char`, `nchar`) is read, its value is blank padded to the same length as the host variable. When a variable length character host variable (`varchar`, `nchar varying`) is read, no blank padding is performed.

When retrieving a value shorter than a fixed length character array host variable, the host variable will be padded with blanks (and null terminated). When retrieving a value to a variable length character host variable, the variable will not be blank padded, just null terminated.

When retrieving binary data into a character array there is no padding or termination of the binary string, so all the character array elements have significance. The character array must, therefore, be declared with exactly the same length as the binary data.

If a numeric type conversion is done when retrieving a value to a fixed length character host variable, the data will be right justified. When type conversion is done when retrieving a value to a variable length character host variable, the data will be left justified.

Example

```
char cstr[9];
VARCHAR vstr[9];
```

retrieving the value `'abc '` will give the following result:

```
cstr = 'abc     ' /* blankpadded to eight characters */
vstr = 'abc '     /* the same length as the value */
```

retrieving the value 123, the values will be as:

```
cstr = '     123' /* right justified */
vstr = '123'      /* left justified */
```

See the *Mimer SQL Reference Manual, Chapter 6, Special Characters*, for a further discussion of different character string assignments.

Preprocessor Output Format

Output from the ESQL/C preprocessor retains SQL statements from the original source code as comments. Comments on the same line as SQL statements are retained as 'comments within comments', marked by the delimiters `/+` and `+/`.

The preprocessed code is structured to reflect the structuring of the original source code.

The use of the `#line` directive will ensure that any information from the C compiler will correctly reference line numbers in the original source code. It will also help a debugger correctly coordinate display of source lines in the original source file and the generated C file. Refer to information on running ESQL for the platform you are using for details on how to get `#line` directives.

Scope Rules

Host variables follow the same scope rules as ordinary variables in C. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for C is the same as a file (including included files).

ESQL in COBOL Programs

Mimer SQL supports ESQL for COBOL following the COBOL-85 ANSI standard.

SQL Statement Format

The following sections discuss the SQL statement format.

Statement Delimiters

SQL statements are identified by the leading delimiter `EXEC SQL` and terminated by `END-EXEC`.

SQL statements are treated exactly as ordinary COBOL statements with regard to the use of an ending period to mark the end of a COBOL sentence. Any valid COBOL punctuation may be placed after the `END-EXEC` terminator.

Examples:

```
EXEC SQL DELETE FROM countries END-EXEC.

IF SQLSTATE NOT = "02000" THEN
            EXEC SQL COMMIT END-EXEC
ELSE
            EXEC SQL ROLLBACK END-EXEC.
```

Margins

Statements (including delimiters) may be written anywhere between positions 8 and 72 inclusive.

Line Continuation

Line continuation rules for SQL statements are the same as those for ordinary COBOL statements.

If a string constant within an SQL statement is divided over several lines, the first non-blank character on the continuation line must be a string delimiter. There is no terminating string delimiter at the end of the line preceding the continuation line.

Example

```
EXEC SQL SELECT CODE, CURRENCY
            FROM MIMER_STORE.CURRENCIES
            WHERE CODE LIKE :CURRENCY-CODE END-EXEC.
EXEC SQL COMMENT ON TABLE CURRENCIES IS
                    'Holds currency
-                   ' details' END-EXEC.
```

An alternative way to break a character string constant over several lines, is to use a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. `<TAB>, <LF>, <VT>, <FF>, <CR> or <SP>`), to join two or more substrings.

Each substring must be separately enclosed in delimiters.

```
EXEC SQL COMMENT ON TABLE CURRENCIES IS
                    'Holds currency'<CR>
                    ' details' END-EXEC.
```

Comments

Comment lines, marked by an asterisk (*) in position 7, may be written within SQL statements. The whole line following a comment mark is treated as a comment.

Debugging lines and page eject lines (marked by D and / respectively in position 7) are treated as comments by the preprocessor.

Special Characters

The delimiters in SQL are single quotation marks (') for string constants and double quotation marks (") for delimited identifiers. This is contrary to the default COBOL string delimiter usage.

```
EXEC SQL INSERT INTO "tablename" VALUES ('text string') END-EXEC.
```

Observe that the minus sign (-) is valid in variable names in COBOL. All arithmetic expressions using this operator should have at least one space separating the operands from the operator. For example:

:A - B means: variable called A minus column B

:A-B means: variable called A-B

Restrictions

The following restrictions apply specifically to COBOL:

- END-EXEC is a keyword reserved to SQL.
- COBOL figurative constants (such as ZERO and SPACE) may not be used as constants in SQL statements.

Host Variables in COBOL

The following sections discuss declarations, SQL data type correspondence, preprocessor output format and value assignments.

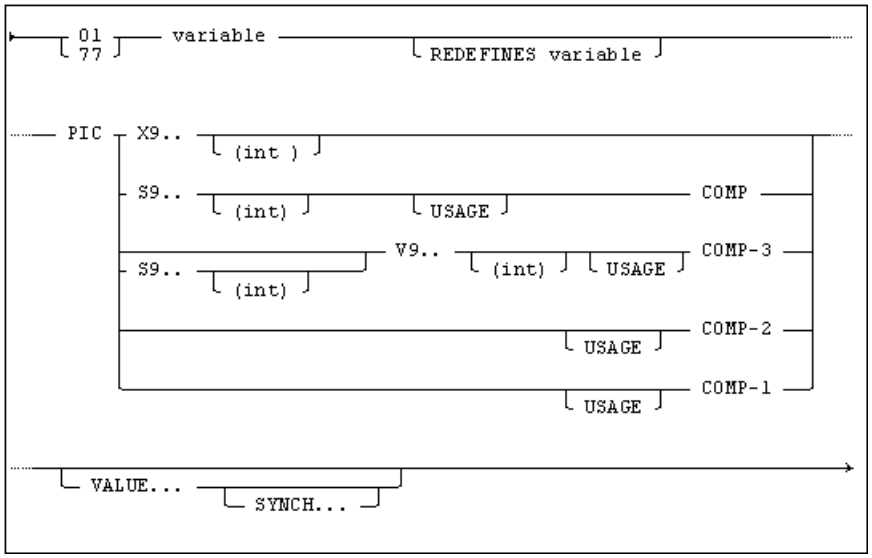
Declarations

Host variables used in SQL statements must be declared within the SQL DECLARE SECTION, delimited by the statements BEGIN DECLARE SECTION and END DECLARE SECTION.

Variables declared within the SQL DECLARE SECTION must conform to the following rules in order to be recognized by the SQL preprocessor:

- variable names must begin with a letter. Within this restriction, any valid COBOL variable name may be used
- host variable structures may not be used
- the specifications JUSTIFIED, BLANK WITH ZERO and OCCURS may not be used
- the data type must be consistent with SQL data types as specified below
- level number 01 or 77 should be used for all variable names that are used in SQL statements. Other levels may be used for program host variables, but they are not recognized by the preprocessor
- FILLER entries are ignored for variables used in SQL statements
- Indicator variables should be declared as PIC S9(4) COMP or PIC S9(9) COMP

A syntax diagram for COBOL variable declarations recognized by the ESQL/COBOL preprocessor is given below. Other declarations are ignored by the preprocessor:



Commas and semicolons may be used in accordance with standard COBOL practice.

The following abbreviations are accepted:

Abbreviation	Full term
PIC	PICTURE or PICTURE IS
USAGE	USAGE or USAGE IS
COMP	COMPUTATIONAL
SYNC	SYNCHRONIZED

Note: The `PIC S9(n)9(m)` formulation is not accepted.

SQL Data Type Correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

Varying-length character string structures may be used in ESQL statements in COBOL programs. In assigning the value of such variables to columns, the current length of the string is used.

The variable name used in SQL statements is the name of the structure (level 01 declaration), not of the character string element (level 49).

SQL data type	COBOL data declaration	Comments
SMALLINT INTEGER BIGINT	01 name PIC S9(n) COMP.	$1 \leq n \leq 9$
DECIMAL	01 name PIC S9(n)V9(m) COMP-3.	$1 \leq n+m \leq 15$
FLOAT DOUBLE PRECISION	01 name COMP-2.	
REAL	01 name COMP-1.	

SQL data type	COBOL data declaration	Comments
CHARACTER VARCHAR DATETIME INTERVAL CLOB	01 name PIC X(n).	$1 \leq n$

Value Assignments

The general rules for conversion of values between compatible but different data types, see the *Mimer SQL Reference Manual, Chapter 6, SQL Syntax Elements*, apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

The first element in a varying-length character string structure is used to store the current length of the character string. When writing to the variable, the first element is updated with the current length of the variable. If the column value is longer than the variable, the value is truncated.

Preprocessor Output Format

Output from the ESQL/COBOL preprocessor retains SQL statements from the original source code as comments. Comments within SQL statements are retained exactly as written. The output follows the ANSI standard for record format, and should be compiled with a COBOL compiler set to accept ANSI standard.

Debugging lines and page eject lines (using `D` and `/` respectively in position 7) remain unchanged after preprocessing.

The preprocessed code is structured to reflect the structuring of the original source code.

Scope Rules

Host variables follows the same scope rules as ordinary variables in COBOL. SQL descriptor names, cursor names and statement names must be unique within the compilation unit. A compilation unit for COBOL is the same as a routine.

ESQL in Fortran Programs

Mimer SQL supports ESQL for ANSI Fortran-90 fixed format.

Source statements must be provided as fixed format, 80 byte records.

SQL Statement Format

The following sections discuss the SQL statement format.

Statement Delimiters

The leading delimiter `EXEC SQL` identifies SQL statements. The end of an SQL statement is marked by the end of the line when the following line does not begin with a continuation character. The Fortran-90 statement delimiter `;` can also be used.

Example

```
EXEC SQL DELETE FROM countries
```

Margins

Statements (including delimiters) may be written anywhere between positions 7 and 72 inclusive.

Line Continuation

Line continuation rules for SQL statements are the same as those for ordinary Fortran statements. The continuation character is any character except space and 0 (zero) in position 6. The Fortran limitation of a maximum of 19 continuation lines per statement does not apply within SQL statements.

For a string constant, a white-space character (ASCII HEX-values 09 - 0D, or 20, i.e. `<TAB>, <LF>, <VT>, <FF>, <CR> or <SP>`), can be used to join two or more substrings. Each substring must be separately enclosed in delimiters.

Examples:

```
      EXEC SQL SELECT CODE, CURRENCY
     +                FROM MIMER_STORE.CURRENCIES
     +                WHERE CODE LIKE :CODE

      EXEC SQL COMMENT ON TABLE CURRENCIES IS
     +                       'Holds currency'<CR>
     +                       ' details'
```

Statement Numbers

Any labeled SQL statement in the source code will generate a CONTINUE statement during preprocessing. Declarative SQL statements used before the first executable SQL statement should not be labeled.

Comments

Comment lines, marked by `*` or `C` in position 1, may be written within SQL statements. The whole line following a comment mark is treated as a comment, and the following line must either be another comment or follow the continuation rules given above.

Note: Lines that are completely blank are not treated as comments by the ESQL/FORTRAN preprocessor. The absence of a continuation character indicates the end of the previous statement, and a completely blank line may be used to structure comments in the output from the preprocessor. See *Preprocessor Output Format* on page *313* for details.

Fortran-90 style comments may also be used, marked by the `!` character (the text between the `!` and the end-of-line is treated as a comment).

Debugging lines (marked with a `D` in position 1) are treated as comments by the preprocessor.

Host Variables

The following sections discuss declarations, SQL data type correspondence, value assignments, preprocessor output format and scope.

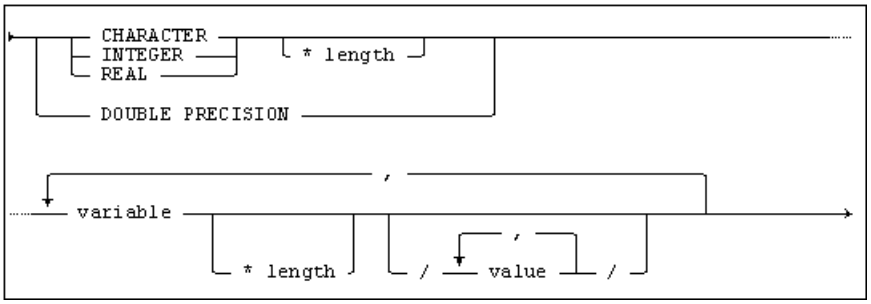
Declarations

Host variables used in SQL statements must be declared within the `SQL DECLARE SECTION`, delimited by the statements `BEGIN DECLARE SECTION` and `END DECLARE SECTION`.

Variables declared within the `SQL DECLARE SECTION` must conform to the following rules in order to be recognized by the SQL preprocessor:

- any valid Fortran variable name may be used.
- variables must be scalar variables (i.e. they may not be elements of vectors or arrays).
- implicit declaration by means of the `IMPLICIT` statement or default typing may not be used.
- Fortran COMPLEX variables may not be used.
- character variables must be declared with a fixed constant length. Expressions and variable length declarations (such as `CHARACTER*(*)`) may not be used.
- indicator variables should be declared as `INTEGER*2` or `INTEGER*4`.

A syntax diagram showing the variable declarations recognized by the ESQL/FORTRAN preprocessor is given below:



The data type declaration must be separated from the variable name by at least one space (which is not required in Fortran declarations outside the `SQL DECLARE SECTION`).

Thus the declaration:

```
INTEGER*2A
```

is not recognized. The required formulation is:

```
INTEGER*2 A
```

Lists of variables following a single default data type declaration are accepted. Any declarations in a list that are not valid in SQL contexts are ignored by the preprocessor. Thus, the following statement declares variables `A` and `D` as `INTEGER*4` and `B` as `INTEGER*2` for use in SQL statements, while the array `C` is ignored:

```
INTEGER*4    A, B*2, C(10), D
```

SQL Data Type Correspondence

Valid host data types are listed below for each of the data types used in SQL statements.

SQL data type	Fortran data declaration
SMALLINT INTEGER BIGINT	INTEGER*2 INTEGER*4 INTEGER*8
DECIMAL	REAL*4
FLOAT DOUBLE PRECISION REAL	REAL*8 DOUBLE PRECISION
CHARACTER VARCHAR DATETIME INTERVAL CLOB	CHARACTER*n

The following additional points should be noted:

- Fortran does not support `DECIMAL` data types. A string of digits including a decimal point is interpreted as a `REAL` constant in Fortran. Exponential notation should always be used to specify floating point values in SQL statements.
- `DOUBLE PRECISION` constants may be written with a `D` as the exponent marker in Fortran (e.g. `1.23D+02`). The only permissible exponent marker within SQL statements is E (e.g. `1.23E+02`).

Value Assignments

The general rules for conversion of values between compatible but different data types, see of the *Mimer SQL Reference Manual*, apply to the transfer of data between the database and host variables, with the data type correspondence as given in the table above.

Preprocessor Output Format

Output from the ESQL/FORTRAN preprocessor retains SQL statements from the original source code as comments. The output follows the ANSI standard for record format, and should be compiled with a Fortran compiler set to accept ANSI standard. Comments within SQL statements are retained exactly as written.

Completely blank lines between SQL statements and following comments cause the preprocessor to write the comments after the generated SQL call. Otherwise comments immediately following SQL statements are output before the generated call. Debugging lines (using `D` in position 1) remain unchanged after preprocessing.

The preprocessed code is structured to reflect the structuring of the original source code.

Scope Rules

Host variables follows the same scope rules as ordinary variables in Fortran.

SQL descriptor names, cursor names and statement names must be unique within the compilation unit.

A compilation unit for Fortran is the same as a routine.

Appendix B

Return Codes

Mimer SQL returns two kinds of return codes to an application, SQLSTATE and a native error code (aka. SQLCODE). The SQLSTATE variable returns a standardized, general error code, which gives a rough description of the status for the most recently executed SQL statement.

GET DIAGNOSTICS can be called to access the exception information stored in the diagnostics area that applies to the most recently executed SQL statement, see *Run-time Errors* on page *70*.

The symbol <%> in the text of error messages listed in this chapter indicates the location of an identifier inserted at run-time

SQLSTATE Return Codes

SQLSTATE contains a 5-character long return code string that indicates the status of an SQL statement. These return codes are standardized following the established standards. Observe that not all standardized SQLSTATE return codes are used by Mimer SQL.

SQLSTATE values consists of two fields. The class field, which is the first two characters of the string, and the subclass field, which is the terminating three characters of the string.

List of SQLSTATE Values

Class	Subclass	Meaning
00	**000**	**Successful completion**
01	**000**	**Warning**
01	002	- disconnect error
01	003	- null value eliminated in set function
01	004	- string data, right truncation
01	005	- insufficient item descriptor areas
01	006	- privilege not revoked
01	007	- privilege not granted
01	008	- implicit zero-bit padding
01	997	- some statement(s) not altered**

Class	Subclass	Meaning
01	998	- row has been updated**
01	999	- row has been deleted**
01	S01	- error in row
01	S02	- option value changed
01	S05	- cancel treated as close
01	S06	- attempt to fetch before the result set returned the first rowset
01	S07	- fractional truncation
02	**000**	**No data**
07	**000**	**Dynamic SQL error**
07	001	- using clause does not match dynamic parameter specifications
07	002	- using clause does not match target specifications
07	003	- cursor specification cannot be executed
07	004	- using clause required for dynamic parameters
07	005	- prepared statement is not a cursor specification
07	006	- restricted data type attribute violation
07	007	- using clause required for result fields
07	008	- invalid descriptor count
07	009	- invalid descriptor index
07	00F	- invalid DATETIME_INTERVAL_CODE
08	**000**	**Connection exception**
08	001	- client unable to establish connection
08	002	- connection name in use
08	003	- connection does not exist
08	004	- server rejected the connection
08	006	- connection failure
08	S01	- communication link failure
09	**000**	**Triggered action exception**
0A	**000**	**Feature not supported**
0B	**000**	**Invalid transaction initiation**
0K	**000**	**Resignal when handler not active**
0W	**000**	**Prohibited statement encountered during trigger execution**

Class	Subclass	Meaning
21	**000**	**Cardinality violation**
21	S01	- insert value list does not match column list
21	S02	- degree of derived table does not match column list
22	**000**	**Data exception**
22	001	- string data, right truncation
22	002	- null value, no indicator parameter
22	003	- numeric value out of range
22	005	- error in assignment
22	006	- invalid interval format
22	007	- invalid datetime format
22	008	- datetime field overflow
22	011	- substring error
22	012	- division by zero
22	015	- interval field overflow
22	018	- invalid character value for cast
22	019	- invalid escape character
22	023	- invalid parameter value
22	024	- unterminated C string
22	025	- invalid escape sequence
22	026	- string data, length mismatch
22	027	- trim error
22	029	- noncharacter in UCS string
23	**000**	**Integrity constraint violation**
24	**000**	**Invalid cursor state**
25	**000**	**Invalid transaction state**
25	S03	- transaction is rolled back
26	**000**	**Invalid SQL statement name**
27	**000**	**Triggered data change violation**
28	**000**	**Invalid authorization specification**
2E	**000**	**Invalid connection name**
2F	**000**	**SQL routine exception**

Class	Subclass	Meaning
2F	003	- prohibited SQL-statement attempted
2F	005	- function executed no return statement
33	**000**	**Invalid SQL descriptor name**
34	**000**	**Invalid cursor name**
35	**000**	**Invalid condition number**
37	**000**	**Syntax error or access violation in PREPARE or EXECUTE IMMEDIATE ***
3C	**000**	**Ambiguous cursor name**
40	000	- transaction rollback
40	001	- serialization failure
42	**000**	**Syntax error or access rule violation**
42	S01	- base table or view already exists
42	S02	- base table or view not found
42	S11	- index already exists
42	S21	- column already exists
42	S22	- column not found
44	**000**	**WITH CHECK OPTION violation**
45	**000**	**Unhandled user-defined exception**
HY	**000**	**General error**
HY	001	- memory allocation error
HY	003	- invalid application buffer type
HY	004	- invalid SQL data type
HY	007	- associated statement is not prepared
HY	008	- operation canceled
HY	009	- invalid use of null pointer
HY	010	- function sequence error
HY	011	- attribute cannot be set now
HY	012	- invalid transaction operation code
HY	013	- memory management error
HY	014	- limit on the number of handles exceeded
HY	015	- no cursor name available
HY	016	- cannot modify an implementation row descriptor

Class	Subclass	Meaning
HY	017	- invalid use of an automatically allocated descriptor handle
HY	018	- server declined cancel requestSQLCancel
HY	019	- non-character and non-binary data sent in pieces
HY	020	- attempt to concatenate a null value
HY	021	- inconsistent descriptor information
HY	024	- invalid attribute value
HY	090	- invalid string or buffer length
HY	091	- invalid descriptor field identifier
HY	092	- invalid attribute/option identifier
HY	093	- invalid parameter number
HY	095	- function type out of range
HY	096	- invalid information type
HY	097	- column type out of range
HY	098	- scope type out of range
HY	099	- nullable type out of range
HY	100	- uniqueness option type out of range
HY	101	- accuracy option type out of range
HY	102	- table type out of range
HY	103	- invalid retrieval code
HY	104	- invalid precision or scale value
HY	105	- invalid parameter type
HY	106	- fetch type out of range
HY	107	- row value out of range
HY	108	- concurrency option out of range
HY	109	- invalid cursor position
HY	110	- invalid driver completion
HY	111	- invalid bookmark value
HY	C00	- optional feature not implemented
HY	T00	- timeout expired
HY	T01	- connection timeout expired
IM	**000**	**ODBC specific return codes**

Class	Subclass	Meaning
IM	001	- driver does not support this function
IM	008	- dialog failed
S0	**000**	**ODBC 2.0 codes ***
S0	001	- base table or view already exists *
S0	002	- base table not found *
S0	011	- index already exists *
S0	012	- index not found *
S0	021	- column already exists *
S0	022	- column not found *
S1	**000**	**General error**
S1	001	- memory allocation failure
S1	002	- invalid column number
S1	003	- program type out of range
S1	004	- SQL data type out of range
S1	008	- operation canceled
S1	009	- invalid argument value
S1	010	- function sequence error
S1	012	- invalid transaction operation code specified
S1	090	- invalid string or buffer length
S1	091	- descriptor type out of range
S1	092	- option type out of range
S1	093	- invalid parameter number
S1	095	- function type out of range
S1	096	- information type out of range
S1	097	- column type out of range
S1	098	- scope type out of range
S1	099	- nullable type out of range
S1	100	- uniqueness option out of range
S1	101	- accuracy option type out of range
S1	102	- table type out of range
S1	105	- direction option out of range

Class	Subclass	Meaning
S1	106	- fetch type out of range
S1	107	- row value out of range
S1	108	- concurrency option out of range
S1	109	- invalid cursor position
S1	C00	- driver not capable
S1	T00	- timeout expired

* Return code will only be returned to ODBC applications executing in ODBC 2.0 mode.
'** Mimer SQL specific SQLSTATE code.

Native Mimer SQL Return Codes

Here the native Mimer SQL return codes are listed together with the associated text message. See *Run-time Errors* on page *70* for details on how to retrieve this information after an exception has been raised.

Sometimes the deprecated term `SQLCODE` is used when referring to the native return code.

The codes are grouped according to function as follows:

Code numbers	Functional group
> 0	*Warnings and Messages*
= 0	Success
100	No data
-100 to -999	*ODBC Errors and Warnings*
-10000 to -10999	*Data-dependent Errors*
-11000 to -11999	*Limits Exceeded*
-12000 to -12999	*SQL Statement Errors*
-14000 to -14999	*Program-dependent Errors*
-16000 to -16999	*Databank and Table Errors*
-18000 to -18999	*Miscellaneous Errors*
-19000 to -19999	*Internal Errors*
-21000 to -21999	*Communication Errors*
-22000 to -22999	*JDBC Errors*
-23000 to -23999	*Mimload Errors*
-24000 to -24999	*Mimer SQL C API Return Codes*

Corrective action is given in general terms for each group of codes. When reporting errors to Mimer support, make sure you include the native Mimer SQL return code.

Warnings and Messages

No corrective action is normally required for internal Mimer SQL return code values greater than zero.

Number	Explanation
53	Null values eliminated in set function
54	Character string was truncated
55	Insufficient item descriptor areas
56	Privilege not revoked
57	Privilege not granted
58	Zero bits were added to the binary string
90	Login failure
91	Soft enter performed
92	No cursor state was saved on stack
94	Message text not found
100	Row not found for FETCH, UPDATE or DELETE, or the result of a query is an empty table
100	No data - Item number is greater than the value of count
997	It was not possible to alter all executable statements
998	The row has been updated by an update where current statement for this cursor
999	The row has been deleted by a delete where current statement for this cursor

ODBC Errors and Warnings

These errors occur when ODBC calls to Mimer SQL fail for some reason.

Number	Explanation
-100	Illegal sequence
-101	Out of memory
-102	Option out of range
-103	Function not supported
-104	Connection not open
-105	Connection in use
-106	Invalid argument value
-107	Invalid transaction operation code

Number	Explanation
-108	Internal network buffer overflow
-109	Invalid C data type
-110	Invalid SQL data type
-111	Bad address
-112	Function already active
-113	Operation canceled
-114	Wrong number of parameters
-115	Use ODBC function SQLEndTran (or SQLTransact for ODBC 2 applications) to commit or rollback transaction
-116	Statement is not in a prepared state
-117	Invalid transaction state
-118	Unknown statement type
-119	Server data type not supported by client
-120	Unknown data type
-121	Invalid buffer length
-122	String data truncated
-123	Numeric data truncated
-124	Numeric value out of range
-125	Invalid numeric value
-126	Bad parameter passed to numeric package
-127	Invalid column number
-128	Database name mandatory
-129	Connect dialog failed
-130	Data truncated
-131	Invalid connection string attribute
-132	Invalid cursor state
-133	Invalid parameter number
-134	Descriptor type out of range
-135	Invalid type passed to DICOA3
-136	Function type out of range
-137	Invalid cursor name
-138	Duplicate cursor name

Number	Explanation
-139	Cursor hash table corrupt
-140	ODBC database control block chain corrupt
-141	Option type out of range
-142	Option value not supported
-143	Option not supported
-144	Invalid row or keyset size
-145	Invalid concurrency option
-146	Invalid fetch type
-147	Not a scrollable cursor
-148	Row position out of range
-149	Only one SQLPutData for fixed length parameter
-150	SQLPutData does not support block cursors
-151	Driver not capable
-152	Table type out of range
-153	Invalid string length
-154	Data type out of range
-155	Syntax error found in escape clause
-156	DDO buffer overflow
-157	Uniqueness option type out of range
-158	Accuracy option type out of range
-159	Column type out of range
-160	Scope type out of range
-161	Nullable type out of range
-162	Internal type mismatch
-163	Conversion between data types not supported
-164	Invalid date, time, or timestamp
-165	Restricted data type attribute violation
-166	Date, time, or timestamp data truncated
-167	Database has not been configured. Run Configure Mimer 7.1
-168	Translated native SQL string was truncated
-169	ODBC extension DATE, TIME or TIMESTAMP is not supported

Number	Explanation
-170	ODBC extension OUTER JOIN is not supported
-171	ODBC extension for procedure invocation is not supported
-172	Unrecognized first word in escape clause, expected 'CALL','FN','OJ','D','T' or 'TS'
-173	This server version does not support the used scalar function
-174	Unrecognized scalar function found in escape clause
-175	Argument missing in scalar function
-176	Too many arguments in scalar function
-177	Syntax error, incomplete escape clause
-178	Syntax error, unmatched apostrophe in string literal
-179	Syntax error, unmatched quote in delimited identifier
-180	Invalid data type specified in scalar function CONVERT
-181	Information type out of range
-182	Parameter type may only be used with procedures
-183	Parameter type out of range
-184	Update and delete where current fully supported (not simulated)
-185	Option value changed
-186	Static scrollable cursor used instead of keyset or dynamic cursor
-187	Error in row, please check next error code
-188	Cancel treated as FreeStatement/CLOSE
-189	Attempt to fetch before the result set returned the first rowset
-190	Invalid cursor position
-191	Unknown first parameter in scalar function TIMESTAMPADD
-192	Unknown first parameter in scalar function TIMESTAMPDIFF
-193	Bad parameter passed to datetime package
-194	Out of critical section objects
-196	Invalid attribute option identifier
-197	Attribute cannot be set now
-198	General error
-199	Invalid use of an automatically allocated descriptor
-200	Invalid descriptor field identifier
-201	Invalid descriptor index

Number	Explanation
-202	Associated statement not prepared
-203	Interval second fraction truncated
-204	Interval truncation error
-205	Interval convert error
-206	Year to month interval cannot be converted to a numeric value because it is not a single field
-207	Interval cannot be converted to a numeric value because it is not a single field
-208	Invalid interval literal
-209	Interval leading field truncation error
-210	Interval trailing field truncated
-211	Binary data truncated
-212	Binary truncation error
-213	Binary length invalid
-214	Binary data invalid
-215	Binary not supported
-216	Inconsistent descriptor information
-217	Cannot modify IRD
-218	Invalid use of null pointer
-219	Character data not hexadecimal
-220	Internal error
-221	Significant parts of datetime/interval string truncated
-222	Interval field truncation
-223	String data truncated
-224	Binary data truncated
-225	String data truncated
-226	Binary data truncated
-227	Internal error - Must put blob separately
-228	Unicode string data conversion error
-229	LOB data larger than specified in SQL_LEN_DATA_AT_EXEC(x)
-230	Distributed transaction mode is active, but no transaction enlist has been performed. May be due to early transaction abort or illegal sequence of calls in application.

Number	Explanation
-232	Wide character data types not supported by server
-233	SQL_LEN_DATA_AT_EXEC(n) only allowed for long data types
-235	Maximum escape clause nesting depth reached
-236	The character string '%' could not be casted to an %
-237	An SQL % interval cannot be casted to a %
-238	A non-fractional part of a TIME or a TIMESTAMP was truncated
-239	The % character string '%' could not be casted to an %
-300	Failed to read dictionary
-801	Error loading library XOLEHLP.DLL or locating entry point to DtcGetTransactionManager in XOLEHLP
-801	Pending transaction, commit or rollback before exit
-802	Unable to retrieve MSDTC transaction object (IID_IDtcToXaHelperSinglePipe)
-802	Invalid transaction number, must be between 1 and <%>
-803	Unable to retrieve MSDTC resource manager cookie
-803	Server version and BSQL version must be the same when using READLOG
-804	Unable to translate MSDTC transaction id to an XA transaction id (XID)
-805	Unable to perform distributed transaction enlist
-806	Invalid sequence of calls within ODBC to MSDTC interface routines
-807	Error initiating transaction object

Data-dependent Errors

These errors arise when an SQL statement cannot be executed correctly because of the data content of variables, expressions, and so on in the statement. The appropriate corrective action is determined by the nature of the error and the specific context in the application program.

Number	Explanation
-10001	Transaction aborted due to conflict with other transaction
-10002	Transaction aborted due to conflict with in-doubt transaction. Do not retry transaction until in-doubt state resolved.
-10003	Transaction aborted due to a preceding problem with databanks or resources
-10004	Current transaction has been invalidated by a conflicting system administration statement

Number	Explanation
-10100	PRIMARY KEY constraint violated, attempt to insert null value in table <%>, column <%>
-10101	PRIMARY KEY constraint violated, attempt to insert duplicate key in table <%>
-10102	Domain constraint <%> violated for table <%>, column <%>
-10103	Table constraint <%> violated for table <%>
-10104	View constraint violation
-10105	Referential constraint <%> violated INSERT/UPDATE operation not valid for table <%>
-10106	INSERT/UPDATE operation not valid for table <%> UPDATE/DELETE operation not valid for table <%>
-10107	The result of a subquery or select into is more than one row
-10108	Result of SELECT INTO or EXECUTE INTO statement is a table of more than one row
-10109	Data type constraint violation (numeric value out of range)
-10110	UNIQUE constraint <%> violated for table <%>
-10111	Domain constraint <%> violated for CAST
-10112	Deferred referential constraint <%> violated, INSERT/UPDATE operation not valid for table <%>
-10113	Deferred referential constraint <%> violated, UPDATE/DELETE operation not valid for table <%>
-10114	The referential action for the constraint <%> would cause multiple updates on the table <%>
-10199	Host variable type packed decimal is not supported
-10200	Reserved numeric operand found during data type conversion
-10201	Length error or incorrect value found during data type conversion
-10202	Division by zero attempted
-10203	Negative overflow occurred during data type conversion
-10204	Positive overflow occurred during data type conversion
-10205	Loss of significance occurred during data type conversion
-10207	Undefined value found during data type conversion
-10208	Restricted data type attribute violation
-10210	Error in assignment
-10211	Undefined value found during data type conversion

Number	Explanation
-10212	Overflow occurred during data type conversion
-10221	The null value cannot be assigned to a host variable because no indicator variable is specified
-10222	Null not allowed for item descriptor area
-10250	Data type not supported
-10301	Loss of significance occurred in arithmetic operation
-10302	Positive overflow occurred in arithmetic operation
-10303	Negative overflow occurred in arithmetic operation
-10304	Division by zero attempted
-10305	Bad parameter encountered in arithmetic operation
-10306	Invalid input for numeric function
-10307	A corrupt numeric value was encountered
-10308	Binary data truncated
-10309	The binary strings are of unequal length
-10310	Invalid character value for CAST
-10311	String data truncated
-10312	Numeric value out of range
-10313	Illegal (negative) substring length
-10314	Like pattern escape character not followed by underscore or percent character
-10315	Length of like pattern escape character is not equal to 1
-10316	Data truncated
-10317	Invalid hexadecimal string
-10318	Invalid regular expression <%>
-10319	The search string used in a BEGINS_WORD or MATCH_WORD invocation contains invalid characters. Only characters that can be used in an identifier are allowed. More information can be found in the section on identifiers in the Mimer SQL Reference Manual.
-10321	Datetime loss of significance
-10322	Datetime positive overflow
-10323	Datetime negative overflow
-10325	Bad parameter encountered in datetime operation
-10326	Datetime illegal operand

Number	Explanation
-10327	Invalid datetime value
-10328	Datetime subtype mismatch
-10329	Invalid datetime format
-10331	Interval loss of significance
-10332	Interval positive overflow
-10333	Interval negative overflow
-10335	Bad parameter encountered in interval operation
-10336	Interval illegal operand
-10337	Invalid interval value
-10338	Interval subtype mismatch
-10339	Invalid interval format
-10340	Length error for large object
-10341	Cannot set attribute value for UDT instance as it is null
-10351	Character string contains non-ascii character
-10352	Character string contains illegal Unicode character
-10354	Bad parameter encountered in Unicode conversion routine
-10355	Bad length parameter encountered in Unicode conversion routine
-10356	Character string not aligned on proper byte boundary
-10361	The used collation cannot be loaded
-10371	Coordinate value out of range
-10372	Latitude value out of range
-10373	Longitude value out of range
-10380	Invalid value in fetch first clause, must be larger than zero
-10381	Invalid value in fetch offset clause, must be positive
-10390	The builtin.uuid character representation is invalid. The expected value is a string in the format HHHHHHHH-HHHH-HHHH-HHHH-HHHHHHHHHHHH, where each H is a hexadecimal value specified by using the symbols 0-9 and A-F.
-10401	Illegal character found when converting from Unicode to ASCII
-10402	Illegal character found when converting from ASCII to Unicode
-10403	The argument to the UNICODE_CHAR function is an invalid Unicode value
-10451	Character or binary string too long

Number	Explanation
-10601	Invalid value for field of item descriptor area
-10602	Invalid datetime or interval code
-10603	PRECISION field of SQL descriptor cannot be set on non-numeric data types
-10701	Invalid LEVEL value in item descriptor area

Limits Exceeded

These errors arise when internal limits in the Mimer SQL system are exceeded. Some of the limitations are determined by installation-specific parameters, while others are fixed by Mimer SQL. In general, errors of this nature require either re-installation of the system with extended limitations or modification of the application program to reduce the system demands.

Contact your Mimer representative if you have difficulty avoiding errors of this nature.

Number	Explanation
-11001	Dynamic storage area exhausted in host level interface (DYNDE3)
-11002	Internal DB dynamic storage area (SQLPOOL) exhausted
-11011	Internal storage (SQLPOOL) for like pattern exhausted
-11012	Transaction list exhausted
-11013	Too many databanks referenced in statement (max 30)
-11014	Too many databanks active in transaction
-11015	The allocated number of realtime tasks in the server has been exhausted
-11047	The maximum number of recursive invocations has been exceeded
-11100	Internal limit exceeded: query stack
-11101	Internal limit exceeded: scan stack
-11102	Internal limit exceeded: generation stack
-11103	Internal limit exceeded: table descriptor list
-11105	Internal limit exceeded: patch table
-11106	Internal limit exceeded: label table
-11107	Internal limit exceeded: traversal stack
-11108	Internal limit exceeded: sco list
-11109	Internal limit exceeded: boolean stack
-11111	Internal limit exceeded: semantic stack
-11112	Internal limit exceeded: working storage

Number	Explanation
-11113	Internal limit exceeded: statement too complex
-11114	Required temporary table row length is <%>, only <%> is possible
-11115	Internal limit exceeded: restriction group pool
-11118	Internal limit exceeded: scan queue

SQL Statement Errors

These errors arise from syntactic or semantic errors in SQL statements. In general, syntactic errors in ESQL programs are detected by the preprocessor, so errors cannot arise at run-time.

Dynamically submitted SQL statements are however parsed at run-time, and the syntax error codes are returned after attempting PREPARE for a syntactically incorrect source statement.

Semantic errors can arise at run-time from both dynamic and static SQL statements.

Number	Explanation
-12001	Too many errors, error collection terminated
-12101	Syntax error, <%> assumed missing
-12102	Syntax error, <%> ignored
-12103	Syntax error, <%> assumed to mean <%>
-12104	Invalid construction
-12106	Internal parser error, analysis aborted
-12107	Syntax analysis resumed here
-12108	Multiple statements not allowed
-12109	Character string literal contains non-ascii character, use nchar literal instead (i.e. N'text')
-12110	Delimited identifier contains non-ascii character
-12111	Delimited identifier contains invalid character
-12112	Delimited identifier contains illegal Unicode character
-12116	The table <%> does not have a generated primary key
-12117	The internal column MIMER_ROWID cannot be altered
-12119	Password string contains non-ascii character
-12120	Table name too long
-12121	String literal too long
-12122	Numeric literal too long

Number	Explanation
-12123	Invalid password string
-12124	<%> is an invalid hexadecimal literal
-12125	Reserved word may not be used as an identifier
-12126	Invalid name
-12127	Character string literal contains illegal Unicode character
-12128	Result of concatenation too long
-12129	Table definition does not include any column specification
-12131	Table definition includes more than one PRIMARY KEY specification
-12132	Only one column allowed in column list
-12152	<%> not allowed in EXECUTE mode
-12153	It is not possible to set a cardinality for a constraint or index if the cardinality for the table, to which it belongs, is null
-12154	User name too long
-12155	Cardinality for a constraint or index must be less or equal to the cardinality of the table to which it belongs
-12156	Column name too long
-12157	Synonym name too long
-12158	Correlation name too long
-12159	Cursor name too long
-12160	Databank name too long
-12161	Shadow name too long
-12162	Host variable name too long
-12163	File name too long
-12164	Label name too long
-12165	Index name too long
-12166	Object name too long
-12167	View name too long
-12168	Domain name too long
-12169	Too many identifier names given
-12174	Syntax error in escape clause, expecting comma before PRODUCT or CONFORMANCE specification
-12175	Syntax error in escape clause, invalid CONFORMANCE specification

Number	Explanation
-12176	Syntax error in escape clause, invalid YEAR specification
-12177	Syntax error in escape clause, invalid PRODUCT specification
-12178	Syntax error in escape clause, invalid VENDOR specification
-12179	Syntax error in escape clause, expecting VENDOR or YEAR after '--(*'
-12180	Syntax error, unexpected token '*)--'
-12181	Syntax error in escape clause, terminating '*)--' missing
-12182	Syntax error in escape clause
-12200	Table <%> not found, table does not exist or no access privilege
-12201	Table reference <%> is ambiguous
-12202	<%> is not a column of an inserted table, updated table or any table identified in a FROM clause
-12203	<%> is neither an object table of an INSERT, UPDATE or DELETE statement, nor specified in a FROM clause
-12204	Column reference <%> ambiguous
-12205	Column <%> not referenced in GROUP BY clause
-12207	DISTINCT specified more than once in a subselect
-12208	SELECT clause of a subquery specifies more than one column
-12209	Column <%> identified in HAVING clause but not included in GROUP BY clause
-12210	Operand of set function includes a set function
-12211	NEXT VALUE of sequence not allowed in ORDER BY clause
-12212	Operand of set function includes a correlated reference specified in an expression
-12213	Set function not specified in a SELECT clause or HAVING clause
-12214	Invalid data type used, expected type is <%>
-12215	Operand not of <%> type
-12216	Operands are not comparable
-12217	Set function containing DISTINCT may not be specified within an expression
-12218	Constant expression not allowed in ORDER BY clause
-12219	The length of the trim character expression is not one
-12220	Expression must be a column
-12221	SELECT clause contains both column expressions and set function expressions

Number	Explanation
-12222	ORDER BY expression contains no valid column reference
-12223	ORDER BY clause invalid because it includes a column name that is not part of the result table
-12224	ORDER BY clause invalid because it includes an integer which does not identify a column of the result table
-12225	The ORDER BY clause is invalid because it includes duplicate column references
-12226	Set function argument not bound in HAVING context
-12227	Duplicate column reference in FOR UPDATE OF clause
-12228	ORDER BY clause must contain only SELECT list items when DISTINCT, GROUP BY or UNION is specified
-12229	Set function not allowed in ORDER BY clause
-12230	Invalid numeric literal
-12231	Update or insert value is null, but the object column <%> cannot contain NULL values
-12232	Insert value must be a constant expression or NULL
-12233	The number of insert values is not the same as the number of object columns
-12234	Insert value not compatible with the column <%>'s <%> data type
-12235	Subquery not allowed in ORDER BY clause
-12236	Column name <%> does not identify a unique column of the result table
-12237	User-defined type <%> not found, does not exist or no usage privilege
-12238	Column <%> cannot be updated because it is derived from a set function or expression
-12239	The use of NULL in a SELECT clause is only allowed in a UNION
-12240	Statement contains too many table references
-12241	Domains not allowed in parameter list
-12242	The corresponding columns of the operands of a UNION do not have compatible column descriptions
-12243	Result table contains a column for which the type cannot be determined
-12244	Operands of a <%> do not have the same number of columns
-12245	FOR UPDATE clause may not be specified because the result table cannot be modified
-12246	Column <%> in the FOR UPDATE clause is not part of the identified table or view

Number	Explanation
-12250	A host variable or parameter marker is not allowed in a view definition
-12251	CREATE VIEW statement must include a column list because the SELECT clause contains an expression
-12252	CREATE VIEW statement must include a column list because the SELECT clause contains duplicated column name <%>
-12253	The number of columns specified for the view is not the same as specified by the SELECT clause
-12254	WITH CHECK OPTION cannot be used for the specified view because it cannot be modified
-12258	<%> operation not permitted because the view cannot be modified
-12259	<%> operation not permitted because the joined table cannot be modified
-12260	LOAD operation not permitted, as there is a trigger for insert defined on the table
-12261	INSERT statement not permitted because the object column <%> is derived from an expression
-12262	The type of the parameter marker cannot be determined
-12263	Parameter markers and host variables not allowed in EXECUTE IMMEDIATE environment
-12264	Parameter markers may not be specified in SELECT clause
-12265	Literal or computed value overflow
-12266	Decimal divide operation invalid because the result would have a negative scale
-12267	Duplicate column reference in INSERT column list
-12268	Duplicate column reference in UPDATE set clauses
-12269	Duplicate column name specified in column list
-12270	<%> does not have <%> privilege on object <%>
-12271	Duplicate column reference in GROUP BY clause
-12273	The types of the results of a CASE expression are not type compatible
-12274	The type of the CASE expression result cannot be determined
-12275	At least one result in a CASE expression must be non-null
-12277	Invalid CAST data type specification
-12278	Cannot extract <%> field from data type <%>
-12280	<%> is an invalid datetime literal
-12281	<%> is an invalid interval literal

Number	Explanation
-12282	Invalid interval qualifier
-12283	Invalid use of interval qualifier
-12284	The recursive WITH-clause <%> may only contain a single UNION-clause
-12285	There can only be a single reference to recursive table <%>
-12286	Incorrectly placed reference to recursive table <%>, it may not occur in right side of outer join, in a subquery etc.
-12287	The number of with-clause columns does not match the underlying query select list
-12288	Cycle detected in recursive WITH clause data
-12289	Search and/or cycle-clause may only be used in recursive with-clause
-12290	Locator types may not be passed to client
-12291	ON COMMIT may only be used with temporary tables
-12292	Cannot create index <%>, uniqueness cannot be enforced when IGNORE NULL specified
-12293	Window function <%> requires ORDER BY in OVER clause
-12294	Duplicate window function name <%>
-12295	Cyclic window function name <%>
-12296	Window function with name <%> not found
-12297	Multiple <%> clauses in window function and referenced window <%>
-12300	Syntax error in escape clause
-12301	Translated native SQL string was truncated
-12302	The function <%> is not supported
-12303	Unrecognized first word, <%> in escape clause expected 'CALL', 'FN','OJ', 'D', 'T' or 'TS'
-12304	Unrecognized scalar function <%>
-12305	Invalid data type <%> in function CONVERT
-12306	Syntax error, incomplete escape clause
-12307	Syntax error, unmatched apostrophe in string literal
-12308	Syntax error, unmatched quote in delimited identifier
-12309	Unknown first parameter <%> in scalar function <%>
-12310	Argument missing in scalar function <%>
-12311	Too many arguments in scalar function <%>
-12330	Statement was not a query

Number	Explanation
-12331	Statement was not an UPDATE, DELETE or INSERT
-12500	A databank named <%> already exists (or filename already used)
-12501	Table <%> does not exist
-12502	<%> does not have <%> privilege
-12503	<%> does not have <%> privilege on object <%>
-12504	Statement not allowed within transaction
-12505	<%> is not a USER or PROGRAM ident
-12506	No privilege
-12507	<%> does not have any databank available (see CREATE DATABANK)
-12509	An ident cannot REVOKE a privilege from itself
-12510	An ident cannot GRANT a privilege to itself
-12511	Duplicate column specification
-12512	Invalid type description
-12513	The cascade option would cause a drop of the last column <%> for table <%> in schema <%>
-12514	The value for <%> must be less than <%>
-12515	The value of <%> must be <%> than or equal to the value of <%>
-12516	Qualified column name required
-12517	Object <%> does not exist
-12518	Circular grant of membership between groups not permitted
-12519	Invalid type description, <%> length must be between <%> and <%>
-12520	An ident cannot GRANT a privilege to itself
-12521	Create <%> option <%> specified more than once
-12522	Alter <%> option <%> specified more than once
-12523	Databank <%> does not exist
-12524	READ ONLY option only available for ALTER DATABANK
-12525	Alter table option <%> specified more than once
-12526	Default value for NOT NULL column <%> missing
-12527	Constraints and unique indexes must be specified WITHOUT CHECK when the database is set to AUTOUPGRADE
-12528	It is not allowed to create a shadow when the database is set to AUTOUPGRADE

Number	Explanation
-12529	The database cannot be set to AUTOUPGRADE when there are shadows defined
-12530	Operand not of type <%>
-12531	Operands not comparable
-12532	Invalid option, should be ON or OFF
-12533	Literal or computed value overflow
-12534	Invalid numeric literal
-12535	Invalid identifier, keyword VALUE expected
-12536	Name <%> in PRIMARY KEY specification not recognized as a column name of current table definition
-12537	<%> must be unqualified
-12538	Default value not compatible with domain definition
-12539	Host variable construction illegal in this context
-12540	<%> is not a column of the specified table(s)
-12541	Data pagesize too small for table record length
-12542	Default value is outside the range specified by domain definition
-12543	Index pagesize too small for table key length
-12544	Too many columns specified in <%> statement
-12545	Primary key column <%> may not be updated because the table is in a WORK databank
-12546	Column <%> is not type compatible with the corresponding column of the referenced table
-12547	Number of columns specified in the foreign key is not the same as the number of columns in the primary key of the referenced table
-12548	Column <%> is not collation compatible with the corresponding column of the referenced table
-12549	Databank option may not be changed to WORK since <%> either contains tables with FOREIGN KEY or UNIQUE constraints or sequences
-12550	Table <%> includes a FOREIGN KEY or UNIQUE constraint and may therefore not be created in a databank with WORK option
-12551	Table <%> is in a databank with WORK option and may therefore not be used as referential constraint
-12552	A UNIQUE index or a UNIQUE or FOREIGN KEY constraint cannot be created for table <%> as it is located in a databank with WORK option
-12553	Explicit grant membership on PUBLIC is not permitted

Number	Explanation
-12554	PUBLIC cannot be member of another group
-12555	The option cannot be changed to READ ONLY since <%> contains a table using a current collation
-12556	<%> cannot be shadowed because it is a WORK databank
-12557	Shadow named <%> already exists
-12558	Ident named <%> already exists
-12559	There is no index <%> defined for <%>
-12560	Table, view, synonym, index or constraint named <%> already exists
-12561	User-defined type or domain <%> not found, does not exist or no usage privilege
-12562	Only foreign key constraints can be deferrable
-12563	Shadow <%> does not exist
-12564	Ident <%> does not exist
-12565	Maximum row length exceeded by index or key table
-12566	Maximum row length exceeded by base table
-12567	It is not possible to add a file to the databank <%> as there is a shadow defined for the databank
-12568	EXISTS construction illegal in this context
-12569	ALL or ANY construction illegal in this context
-12570	Set function construction illegal in this context
-12571	Subquery construction illegal in this context
-12572	Too many columns given in FOREIGN KEY clause
-12573	Name <%> in CHECK clause not recognized as a column name of current table definition
-12574	Name <%> in column constraint not recognized as current column name
-12575	Constraints for generated keys cannot be dropped explicitly
-12577	Default value not compatible with column specification
-12579	No such unique constraint in referenced table
-12580	It is not possible to add new columns to the table since the <%> has an implicit reference to all columns in <%> statement. There may be further dependencies.
-12581	Too many columns given in UNIQUE constraint
-12582	UNIQUE constraint equivalent to PRIMARY KEY constraint
-12583	UNIQUE constraint equivalent to previously given UNIQUE constraint

Number	Explanation
-12584	The added constraint is equivalent to a previously defined PRIMARY KEY or UNIQUE constraint
-12585	Name <%> in FOREIGN KEY clause not recognized as a column name of current table definition
-12586	Data pagesize not supported
-12587	Index pagesize not supported
-12588	Compression <%> not supported
-12589	An ident or a schema owning any collation cannot be dropped. Drop collation first.
-12590	Table contains too many columns
-12591	Cannot create unique index
-12592	Dependencies exist, RESTRICT specified
-12593	Column <%> does not exist
-12594	Column <%> cannot be dropped as it is the only column in table
-12595	Column <%> cannot be dropped, dependencies exist
-12596	Default value for column <%> does not exist
-12597	Change of data type is not allowed for a column included in a key or index
-12598	The data type for a column cannot be changed to a domain if that domain has any check constraints
-12599	The proposed data type change is not supported
-12600	Change of data type is not supported for a column used by a view, routine or trigger
-12601	Statement does not support backup of <%>
-12602	The same file name is given for backup and incremental backup
-12603	Database is already OFFLINE
-12604	Database is already ONLINE
-12605	Cannot RESET LOG, because database is ONLINE
-12606	Databank <%> is already OFFLINE
-12607	Databank <%> is already ONLINE
-12608	Cannot RESET LOG, because databank <%> is ONLINE
-12609	Shadow <%> is already OFFLINE
-12610	Shadow <%> is already ONLINE
-12611	Cannot RESET LOG, because shadow <%> is ONLINE

Number	Explanation
-12612	Shadow <%> is already specified
-12613	Cannot set more than one shadow OFFLINE for databank having shadow <%>
-12614	Statistics cannot be updated for <%> because it is a view
-12615	Filename already used by databank or shadow
-12616	Cannot SET DATABASE OFFLINE, because another user is connected
-12617	Cannot SET DATABANK <%> OFFLINE, because the databank is in use
-12618	INCREMENTAL backups can only be used in conjunction with EXCLUSIVE
-12619	The column <%> is not of structured type
-12620	A domain or user-defined type named <%> already exists
-12621	A schema named <%> already exists
-12622	The schema name for the index table must be the same as the schema name for the table
-12623	A PRIMARY KEY constraint for this table has already been defined
-12624	The added PRIMARY KEY or UNIQUE constraint cannot be created as there are duplicates
-12625	The added referential constraint is not fulfilled by existing records
-12626	The added check constraint criteria is not fulfilled by existing records
-12627	One or more specified columns does not exist in table
-12628	Referenced table or column not found
-12629	FOREIGN KEY not referencing a compatible UNIQUE or PRIMARY KEY
-12630	Constraint <%> cannot be dropped, dependencies exist
-12631	PRIMARY KEY or UNIQUE column constraint not valid when records exist
-12632	Collation <%> does not exist
-12633	A collation named <%> already exists
-12634	Invalid collation definition string
-12635	Collation <%> cannot be dropped, used by domain or column
-12636	A collate clause can only be specified for character types
-12637	Expressions with different collating sequences cannot be compared or concatenated

Number	Explanation
-12638	Collation specified in ORDER BY must be the same as used in DISTINCT/UNION
-12639	Column of type large object not allowed in PRIMARY KEY, UNIQUE, FOREIGN KEY or INDEX
-12640	Precompiled statement <%> does not exist or no execute privilege
-12641	A precompiled statement named <%> already exists
-12642	Invalid SQL construction used in CREATE STATEMENT
-12643	Column of data type large object not allowed in this context
-12644	SQL syntax error, only EXECUTE STATEMENT allowed
-12645	Column of data type large object not allowed in a databank having shadow
-12646	Precompiled statement is not scrollable
-12647	Precompiled statement is scroll only
-12648	Not allowed to change table as it is used by a precompiled statement
-12649	Statement <%> was compiled with an earlier version
-12650	Only the creator of a precompiled statement may alter it
-12651	Invalid syntax in statement invocation
-12562	The constructor function <%> can only be used in a NEW invocation
-12653	A foreign key constraint between <%> and <%> using the same columns and referencing the same key already exists
-12654	It is not possible to create a special index on a table that has fixed format
-12655	It is not possible to alter table to fixed format since there is a special index defined on the table
-12656	It is not possible to add a file to the databank <%> since it is removable
-12657	It is not possible to set the databank <%> as removable since it has multiple files
-12660	No column result set
-12661	The data type list does not match any routine
-12662	There is no matching <%> with the specified number of parameters
-12663	Invalid type description, <%> precision must be between <%> and <%>
-12664	Invalid type description, <%> fraction must be between <%> and <%>
-12665	It is not possible to cast data with type <%> to <%>
-12666	It is not possible to assign data with type <%> to <%>
-12667	It is not possible to compare data with types <%> and <%>

Number	Explanation
-12668	It is not possible to concatenate <%> data and <%> data
-12669	It is not possible to use arithmetic operations on data with types <%> and <%>
-12670	Support for structured user-defined types is not included in this version
-12671	Support for read only databanks is not included in this version
-12672	Support for this diagnostic item is not included in this version
-12673	CURRENT_COLLATION not allowed for column in unique constraint
-12674	CURRENT_COLLATION not allowed for column in primary key
-12675	CURRENT_COLLATION not allowed for column in unique index
-12676	Collation <%> may not be used as base since it is binary
-12678	INSERT columns must be specified when CREATE STATEMENT, i.e. INSERT INTO table (column-list) ...
-12679	SELECT * not allowed when CREATE STATEMENT, specify selected columns (at top level)
-12680	An ident may not drop himself
-12681	CASE expression's return values have different collating sequences
-12682	Columns defined as structured user-defined types cannot be used in constraints or indexes
-12683	Invalid combination of sequence attributes
-12684	The column <%> has invalid type for a PINYIN index, must be NCHAR or NCHAR VARYING
-12685	Invalid <%> value specified for column <%>
-12686	Unable to determine result type for expression, use syntax: (datetime1 - datetime2)interval qualifier, for example: (date '2018-03-04' - date'1634-01-06')year(3) to month
-12687	Invalid <%> value specified
-12701	<%> is a reserved word, and cannot be used as the name for a symbol
-12702	<%> is a global variable, and cannot be used as the name for a variable or parameter
-12703	The class <%> is already present in the handler declaration
-12704	The SQLSTATE <%> is already present in the handler declaration
-12705	The condition identifier <%> is already present in the handler declaration
-12706	The condition identifier <%> is already used in another handler in this scope

Number	Explanation
-12707	A condition identifier for the SQLSTATE <%> has already been defined in this scope
-12708	<%> is not a valid SQLSTATE value
-12709	The SQLSTATE <%> associated with the condition identifier <%> is already present in the handler declaration
-12710	An exception handler for the state (<%>) associated with the condition identifier <%> has already been defined in this scope
-12711	An exception handler for the state <%> has already been defined in this scope
-12712	An exception handler for the class <%> has already been defined in this scope
-12713	The default literal is too large for this data type
-12714	The literal value is too large for this data type
-12715	The type of the default value is not compatible with the type of the variable
-12716	A locator can only be declared for a lob type
-12717	Invalid declaration. The maximum precision for this data type is <%>
-12718	The scale cannot exceed the precision
-12719	A lob type can only be used in PSM if it is declared as a locator
-12720	The number of correlation names for <%>, does not match number of result types
-12721	The parameter <%> must be declared as IN as it is defined in a function, method or result set procedure
-12722	The parameter <%> is declared as IN, and cannot be assigned
-12723	The parameter <%> is declared as OUT, and cannot be used in expressions
-12724	<%> cannot be assigned a value directly
-12725	Result set procedures can only be used in cursor declarations
-12726	The formal argument of the routine is IN but the actual argument is OUT
-12727	The formal argument of the routine is OUT but the actual argument is IN
-12728	Literals or expressions cannot be used for OUT parameters
-12729	Return statements are only allowed in result set procedures or functions
-12730	Recursive call to <%>, not allowed
-12731	<%> statement not allowed in result set procedure
-12732	The procedure <%> does not return a result set and cannot be used in a declare cursor for call

Number	Explanation
-12733	RESIGNAL statement only allowed in exception
-12734	Incorrect number of items in return statement, <%> expected
-12735	Incorrect number of parameters, <%> expected when invoking routine <%>
-12736	Invalid type for argument, <%> expected
-12737	Too long name <%>
-12738	Duplicate declaration <%>
-12739	x.y.z names not allowed
-12740	There is no matching start label for the ending label <%>
-12741	The label or routine <%> is not defined
-12742	The label <%> is not defined
-12743	Procedure <%> does not exist or no execute privilege
-12744	The variable <%> is not defined
-12745	The condition identifier <%> is not defined
-12746	The cursor <%> is not defined
-12747	The use of a domain (<%>) is invalid in this context
-12748	Operands are incompatible
-12749	The SQL module <%> already exists
-12750	There already exists a procedure named <%> having the same parameters
-12751	Duplicate parameter <%>
-12752	The procedure <%> is declared in an SQL module, and cannot be dropped directly
-12753	Failed to read data dictionary
-12754	The length of a host variable cannot exceed 32
-12755	Host variables cannot be used within a procedure
-12756	The data type for the parameter marker cannot be determined
-12757	The number of items in the into clause is less than the number of items in the select list
-12758	The number of items in the into clause is greater than the number of items in the select list
-12759	The number of items in the fetch into clause is less than the number of items in the cursor declaration
-12760	The number of items in the fetch into clause is greater than the number of items in the cursor declaration

Number	Explanation
-12761	The cursor <%> is declared as not scrollable, only next is allowed as fetch direction
-12762	The value in a fetch absolute or relative must be an integer
-12763	The argument to fetch absolute must be larger than 0
-12764	A cursor for call is read only and cannot be used in an update or delete where current statement
-12765	The cursor is declared as read only and cannot be used in an update or delete where current statement
-12766	The table name in the <%> statement is not the same as the name used in the cursor declaration
-12767	The column <%> is not specified in the for update list of the cursor declaration
-12768	The size for a data type must be larger than zero
-12769	It is not allowed to declare exception handlers or condition identifiers for the SQLSTATE successful completion ('00000')
-12770	The formal argument of the procedure is declared as INOUT but the actual argument is <%>
-12771	A handler declaration cannot contain both an exception class and SQLSTATE values or condition identifiers
-12772	The procedure does not return a result set and therefore cannot be used with a scroll cursor
-12773	<%> is not a column and is not declared as a variable or parameter
-12774	The label <%> has already been declared
-12775	The statement requires <%> access
-12776	<%> statement is only allowed if the access indication is MODIFIES SQL DATA or READS SQL DATA
-12777	<%> statements are only allowed if the access indication is MODIFIES SQL DATA
-12778	The access indication for a result set procedure must be READS SQL DATA or CONTAINS SQL
-12779	An UNDO exception handler can only be specified in an atomic compound statement
-12780	<%> is an invalid argument for the signal statement, only integer values allowed
-12781	Only assignment and comparison operations allowed
-12782	The column name <%> has already occurred in this row declaration
-12783	The field <%> is not defined in the row data type for the variable <%>

Number	Explanation
-12784	The row data types do not have the same number of fields
-12785	A row data type variable may not be used as a parameter or result type nor in a DML statement
-12786	A parameter specified as RESULT is not allowed in <%>
-12787	If a RESULT parameter is specified the return data type for the SQL invoked function must be a structured user-defined type and the same as the data type for the parameter
-12788	There can only be one RESULT parameter for an SQL invoked function
-12789	The data type for a parameter defined as RESULT must be a structured user-defined type and it must be the same as the data type returned by the SQL invoked function
-12790	The function <%> can not be altered since it is used as an ordering function for the structured user-defined type <%>. The function must be deterministic and the return data type must be a predefined data type excluding large objects.
-12791	The function <%> can not be altered since it is used as an ordering function for the structured user-defined type <%>. The function must be deterministic and the return data type must be an integer.
-12792	The collating sequence for the parameter <%> is different from the collating sequence used in the method specification
-12793	The collating sequence for the return data type is different from the collating sequence used in the method specification
-12794	The schema names for the dropped method and the user-defined type are not equal
-12795	Specifying a value by using the keyword DEFAULT is only allowed in INSERT and UPDATE statements
-12796	Specifying a value by using the keyword NULL is only allowed in INSERT and UPDATE statements or in a cast expression
-12797	It is not allowed to use the builtin.serialize function on a record that is part of another record
-12798	The result of the builtin.deserialize function cannot be assigned to a record that is part of another record
-12800	Functionality not supported: <%>
-12801	Referencing OLD <%> is not allowed if trigger event is INSERT
-12802	Referencing NEW <%> is not allowed if trigger event is DELETE
-12803	The compound statement in a triggered action must be atomic
-12804	Referencing OLD or NEW ROW may only be used if FOR EACH ROW is specified

Number	Explanation
-12805	A column list can only be specified if trigger event is UPDATE
-12806	Duplicate column name in OF list
-12807	It is not allowed to modify the <%> table
-12808	The trigger time for a view must be INSTEAD OF
-12809	The trigger time for a base table cannot be INSTEAD OF
-12810	Only the creator of a table can create a trigger for the table
-12811	It is not allowed to create triggers for tables, located in databanks with WORK option
-12812	Referencing OLD or NEW table is not allowed in FOR EACH ROW triggers
-12813	A trigger named <%> already exists
-12814	Sequences cannot be created in a databank with WORK option
-12815	The OS_USER name already exists for this ident
-12816	The OS_USER does not exists for this ident
-12817	Add or drop OS_USER is only allowed for idents defined as USER
-12818	Drop password is only allowed for idents defined as USER
-12819	The view cannot be used in an Experience server since it uses functionality that is specific to an Engine server
-12820	The statement <%> is invalid because it could not be upgraded from an earlier version of Mimer SQL
-12821	It is not possible to change the name of a parameter when altering a routine
-12822	The alter routine statement would cause the loss of grant option for execute on the routine. This is due to the presence of statements that uses a privilege which is held without grant option. The first reference of this type is <%> privilege on <%> <%>.
-12823	The data type for a parameter declared as RESULT must be the same as the data type in the returns clause
-12824	It is not possible to alter the returns data type to an incompatible data type or to change the number of result items since there are objects using this routine. The first such object is the <%> <%>.
-12825	The return data type for the function <%> can not be changed since there is a relative ordering for the user-defined type <%> using this function. The return type must be an integer type.
-12826	The alter routine is not allowed. If a return statement is present in the routine body a returns clause must also be given.
-12827	The alter operation is invalid since the routine is declared as deterministic but it contains expressions that are not deterministic

Number	Explanation
-12828	A null call clause can not be specified for a procedure
-12829	Routine attributes or specific name for a method can only be changed by altering the method specification. A method specification is altered by using the ALTER TYPE statement.
-12830	There already exists a function named <%> having the same parameters
-12831	Function <%> does not exist or no execute privilege
-12832	The result of the expression is not deterministic while the routine is declared as deterministic
-12833	All privileges used in a trigger must be held with grant option. This is violated as <%> privilege on the <%> <%> is not held with grant option.
-12834	Sequence <%> does not exist, or no privilege
-12835	A sequence named <%> already exists
-12836	The keyword NULL cannot be used <%>
-12837	The operands to an overlaps predicate must be of a row data type with two elements
-12838	The two elements in an operand to the overlaps predicate must either be of the same type or otherwise it shall be possible to add the second value to first value
-12839	<%> is not allowed in a before trigger
-12840	The simple value specification for a get diagnostics statement must be of integer type
-12841	<%> is not allowed in a trigger
-12842	The increment for a sequence must be non zero
-12843	Invalid values for sequence attributes
-12844	Schema <%> does not exist or no privilege
-12845	The schema name for routines in a module definition must be the same as the schema name for the module
-12846	The value for diagnostics size must be positive
-12847	The ident name in an authorization clause must be the same as the current user
-12848	A constraint named <%> already exists
-12849	The function <%> MODIFIES SQL DATA and can thus not be used in a DML statement
-12850	A trigger must be located in the same schema as the table
-12851	A constraint must have the same schema name as the object to which it belongs

Number	Explanation
-12852	The schema name for a routine is not the same as the schema name for the module
-12853	Ident name not allowed as a schema with the same name exists
-12854	A non-deterministic expression is not allowed in a check clause
-12855	Default values with a reference to a sequence combined with a check clause is not allowed in an alter table statement
-12856	References to LOB columns in the NEW table in an instead of trigger is not allowed
-12857	The label <%> cannot be used for an iterate statement
-12858	The record <%> is not compatible with the <%>
-12859	Creation of recursive statements is not supported
-12860	No user-defined type or domain named <%> is defined
-12861	The attribute name <%> is already specified in this user-defined type or in a supertype
-12862	INSTANTIABLE or NOT INSTANTIABLE cannot be specified for a distinct type
-12863	FINAL or NOT FINAL cannot be specified for a distinct type
-12864	A type cannot be FINAL and NOT INSTANTIABLE
-12865	The default value for the attribute is invalid
-12866	There already exists a routine with specific name <%>
-12867	Parameters for functions and methods may not be specified as IN explicitly
-12868	A parameter name for the routine is missing
-12869	The type attribute <%> specified multiple times
-12870	No specification for the method <%> was found
-12871	The parameter type does not match method specification for <%>
-12872	The parameter name <%> does not match the name used in method specification
-12873	A type name must be specified
-12874	A method must be created in the same schema as the type
-12875	A type binding can only be specified for methods
-12876	A structured type must be declared as FINAL or NOT FINAL
-12877	When creating a user-defined type it is not possible to use that type as the data type for an attribute or as a supertype in an under clause

Number	Explanation
-12878	No <%> method specification exists
-12879	The method does not have the same return data type as the method specification
-12880	SELF is not allowed as parameter or variable name in an instance method
-12881	A method named <%> is already defined for this user-defined type
-12882	A method specification named <%> with equal parameters is already defined for this user-defined type or for a subtype or supertype
-12883	The type for the parameter <%> does not match the method specification
-12884	Type user-defined <%> does not exist or no usage privilege
-12885	As the type <%> is declared as NOT INSTANTIABLE it cannot be used when defining columns or variables
-12886	The method specification <%> does not have a returns clause
-12887	The schema name for the type <%> and the method specification are not the same
-12888	<%> is a static method and must be invoked with the syntax typename::method
-12889	<%> is an instance method and must be invoked with the syntax object.method
-12890	There is no method <%> defined for the type, or no execute privilege
-12891	The attribute <%> does not exist in the type <%>
-12892	The subject routine for <%> cannot be determined, there exists multiple routines with the same name. Provide more specific data types
-12893	The method specification for <%> cannot be determined, there exists multiple method specifications with the same number of parameters but none matching the data types for this method definition
-12894	The distinct type cannot be created since there exists a function with the name <%> and one parameter having the same type
-12895	The object <%> has been created implicitly and cannot be altered or dropped
-12896	A constructor method can only be defined for a structured user-defined type
-12897	A constructor method must have the same name as the user-defined type
-12898	A constructor method must have the same return type as the user-defined type, <%> does not fulfill this requirement
-12899	<%> is a constructor method and must be invoked using the new operator
-12900	The attribute name is the same as an existing routine

Number	Explanation
-12901	Source as distinct is only valid for distinct types
-12902	Multiple source as distinct or distinct as source clauses not allowed
-12904	Order full not allowed for state orderings
-12905	There is no EQUALS ordering defined for the type <%>, comparison not allowed
-12906	Not allowed to create or drop ordering for distinct type
-12907	The user-defined type does not have any ordering
-12908	There is already an ordering defined for this type
-12909	<%> orderings are not supported
-12910	A function named EQUALS with the same parameter types already exists
-12911	There is no FULL ordering defined for the type <%>, comparison not allowed
-12912	Distinct as source is only valid for distinct types
-12913	Only method specifications can be dropped with the alter type statement
-12914	CASCADE not allowed for alter type or alter routine
-12915	Cast to and from structured types not allowed
-12916	Not possible to add or drop attributes for a distinct type
-12917	Only the creator of a user-defined type may alter it
-12918	Not allowed to add or drop attribute to this type as it is used as the data type for a column
-12919	The addition of the attribute would cause a circular dependency
-12920	<%> is the last attribute for the type and cannot be dropped
-12921	The cursor <%> is declared in a for statement and cannot be used explicitly
-12922	All items in the select list in a for statement must be named
-12923	Duplicate name <%> in select list in for statement
-12924	All result items for a result set procedure must have a name when used in a for statement
-12925	Duplicate result item name <%> for result set procedure which is not allowed when used in a for statement
-12926	Locators can only be compared with = or <>
-12927	Locators cannot be used in expressions
-12928	The information item <%> has already been specified

Number	Explanation
-12929	<%> is not a constructor function, and cannot be invoked using the new operator
-12930	The constructor function for the user-defined type <%> cannot be created since there already exists a function with the same name and parameters
-12931	The default value for <%> is not within the range of the data type
-12932	The literal value <%> is outside the valid range for the data type of the variable <%>
-12933	The schema for the user-defined type does not match the schema for the method
-12934	As the type <%> is distinct, a new specification is not allowed
-12935	There is no method specification for the type <%>, with the number of parameters used in the create method statement
-12936	Invalid type for argument number <%>, expected <%> when invoking routine <%>
-12937	Invalid type for argument number <%>, expected <%> in operation <%>
-12938	<%> has already been specified
-12939	Ident <%> is not authorized to create external routines
-12940	The host variable <%> is used in multiple contexts where the data types are incompatible
-12941	The host variable <%> will be assigned multiple times
-12942	The host variable <%> has inconsistent use of indicator variables
-12943	The special column MIMER_ROWID cannot be accessed in a before row trigger
-12944	<%> cannot be modified as it is defined as a large object
-12945	The definition for the index <%> contains multiple type clauses which is not allowed
-12946	The column <%> has an invalid data type. If the index type is <%> only <%> is allowed
-12947	The index <%> is defined as unique, which is not allowed for index type <%>
-12948	The data type of <%> is not compatible with the data type of <%>
-12949	<%> is a field in the old row variable in a trigger and cannot be assigned
-12950	It is not allowed to specify size for a locator
-12951	The type <%> is not instantiable
-12952	No UNDER privilege on type <%>

Number	Explanation
-12953	The type <%> is final and cannot be inherited from
-12954	A distinct type cannot have an under clause
-12955	The type <%> is distinct and cannot be inherited from
-12956	A state or relative ordering cannot be defined for an inherited type
-12957	The routine <%> cannot be used as an ordering function. A map ordering function should have one parameter with the same type as the user-defined type for which the ordering is defined and the return type must be a predefined data type. The function must also be deterministic and cannot modify SQL data.
-12958	The routine <%> cannot be used as an ordering function. A relative ordering function should have two parameters with the same type as the user-defined type for which the ordering is defined and the return type must be integer. The function must also be deterministic and cannot modify SQL data.
-12960	The type <%> must be a subtype of <%> (the type of the left operand) when used in a type predicate or treat expression
-12961	The overriding attribute cannot be specified for a method specification defined as static or constructor, it is only allowed for instance method specifications
-12962	There is no matching original method specification for <%> in any supertype of <%>
-12963	The method <%> is an observer or mutator method in a supertype of <%> and cannot be overridden
-12964	The attribute <%> is inherited and cannot be dropped
-12965	A realtime aggregation for this statement already exists
-12966	The statement <%> can not be used for realtime aggregation since it is not of type select;
-12967	The interval value <%> does not match the data type <%>, expected format is <%> with each field in the valid range
-12968	You are not authorized to create an external library
-12969	There already exists a library with the name <%>
-12970	There already exists a library with the name <%> already used by another library
-12971	The interval value <%> is invalid. The maximum value for the <%> field in an <%> is <%>.
-12972	The interval value <%> is too large with regard to the given precision
-12973	The library <%> does not exist or you are not authorized to use it

Number	Explanation
-12974	The value <%> does not match the data type <%>, expected format is <%> with each field in the valid range
-12975	Invalid access mode NO SQL specified for the routine <%>. NO SQL is only allowed for external routines.
-12976	The file <%> is not defined for the databank <%>
-12977	<%> is the last remaining file for the databank <%> and cannot be dropped
-12978	There already exists a file named <%> for the databank <%>
-12979	It is not allowed to add files for the SYSDB databank
-12980	At most 15 files can be specified for a databank
-12981	As the databank <%> has multiple files it is not possible to change a file specific option without specifying a file name. Use the ALTER DATABANK ... ALTER FILE ... statement instead.

Program-dependent Errors

These errors arise as a result of incorrect program construction. In general, corrective action requires revision of the program source code.

Number	Explanation
-14001	Invalid sequence of SQL statements
-14002	SQL statement invalid because the user is not connected
-14003	CONNECT statement invalid because the user is already connected
-14004	System already closed down
-14005	Cannot perform DISCONNECT in a transaction
-14006	Login failure
-14007	The option for the databank cannot be changed since a backup of the databank is active
-14008	The option for the databank cannot be changed since there are shadows defined on the databank
-14009	Operation is not allowed when a transaction is active
-14010	Operation is not allowed in an atomic execution context
-14011	Transaction already started
-14012	Transaction handling required
-14013	No transaction started
-14014	Cannot perform write operations as transaction status is read-only

Number	Explanation
-14015	Commit or rollback statements are not allowed in an atomic execution context
-14016	Select for update is not allowed for a read-only cursor
-14017	Mixing DDL and DML statements in a transaction is not allowed
-14018	Incremental backup not allowed in BACKUP transaction
-14019	Operation not allowed in BACKUP transaction
-14020	START BACKUP command required to perform an online backup
-14021	Cannot perform ENTER or LEAVE in a transaction
-14022	Cannot perform ENTER operation because program level is already active
-14023	No program level entered, cannot perform leave operation
-14024	Session statements are not allowed in a trigger
-14025	START LOAD command required to perform a load operation
-14026	The table name specified in the COMMIT/ROLLBACK LOAD statement does not match the table name (<%>) specified in the START LOAD statement
-14027	Invalid transaction state, held cursor requires same isolation level
-14028	The routine <%> tries to modify SQL data which is not allowed with the current access mode
-14029	The routine <%> tries to read SQL data which is not allowed with the current access mode
-14031	DESCRIBE statement does not identify a prepared statement
-14032	EXECUTE statement does not identify a prepared statement
-14033	PREPARE statement identifies a SELECT statement of an opened cursor
-14034	The cursor is not in a prepared state
-14035	The cursor identified in a FETCH or CLOSE statement is not open
-14036	The cursor cannot be used because its statement name does not identify a prepared SELECT statement
-14037	The cursor identified in the UPDATE or DELETE statement is not open
-14038	UPDATE or DELETE CURRENT statement not allowed for a cursor of a prepared SELECT statement
-14039	Cursor is not scrollable
-14041	The cursor identified in the UPDATE statement is not positioned on a row
-14042	The cursor identified in the DELETE statement is not positioned on a row
-14043	Routine signaled SQLSTATE

Number	Explanation
-14044	Routine signaled a condition
-14045	Prepared statement not a cursor specification
-14046	The statement RESIGNAL was used outside a exception handler
-14047	Duplicate transaction identifier
-14048	Current user not owner of transaction
-14049	Already associated with transaction
-14050	The specified transaction identifier could not be found
-14051	Invalid distributed transaction state
-14052	It is not possible to mix local and distributed transactions
-14053	The transaction identifier is in use by other user
-14054	The requested operation cannot be performed in a distributed transaction
-14061	Cursor has been invalidated by a conflicting system administration statement
-14101	Invalid statement identifier
-14201	Compilation did not yield an executable program
-14202	The cursor identified in an OPEN statement is already open, but not declared as REOPENABLE
-14203	Statement position cannot be saved when temporary tables are used in the query
-14210	Cursor for result set procedure may not be reopenable
-14211	WITH HOLD option is only available for SELECT statements
-14212	The external routine <%> was not found
-14213	Error when executing external routine <%> Error code from routine: <%>
-14214	The number of parameters does not match the external routine
-14215	The data types of the parameters do not match the definition of the external routine
-14216	Invalid databank option, valid options are READ ONLY, WORK, TRANSACTION or LOG
-14217	The global variable <%> has not been assigned
-14218	The global variable <%> which is declared as <%> cannot implicitly be converted to <%>
-14219	Cannot open external library <%> in file <%>, <%>
-14220	The external routine <%> was not found, <%>

Number	Explanation
-14221	Error when executing external routine <%>, <%>
-14222	The version number of external library <%> is not compatible with the database server. Please recompile the library.
-14223	Illegal format character for a parameter in external routine <%>
-14224	The initiation routine in external library <%> did not call init()
-14225	The format character for a parameter does not support the specified database type
-14226	Memory allocation for stream failed
-14227	It is not possible to add more data to this stream since it has been assigned to a routine parameter
-14301	SQLDA contains an invalid data address or indicator variable address
-14302	Invalid address of username or password
-14303	Invalid address
-14311	Illegal statement length given for SQL statement
-14312	Input character string too long
-14313	Like pattern string too long
-14314	Server name, username, password or connection name too long
-14315	Illegal byte length of floating point number
-14316	Unterminated C string, null byte not found
-14318	Username or password too long
-14321	Illegal host variable type
-14322	Illegal host variable type in like pattern string
-14323	Username and password must be fixed length character strings
-14324	Illegal indicator variable type
-14331	The number of provided host variables does not match the number of parameters
-14332	Using clause required for dynamic parameters
-14333	Using clause required for result fields
-14401	Column cannot be updated because it is not identified in the UPDATE clause of the SELECT statement of the cursor
-14402	The table identified in the UPDATE or DELETE statement is not the same as that designated by the cursor
-14403	Cannot describe statement without naming information
-14404	Cannot use update where current for table <%> since the cursor is read-only

Number	Explanation
-14405	Cannot use delete where current for table <%> since the cursor is read-only
-14406	Unexpected statement type encountered in an UPDATE or DELETE statement
-14410	Update or delete where current is not allowed for tables located in a databank with WORK option
-14501	Database connection is not open
-14601	Invalid cursor state
-14611	Using clause does not match dynamic parameters
-14612	Using clause or Into clause does not match target specifications
-14621	Cursor not found
-14622	Ambiguous cursor name
-14623	Invalid cursor name
-14624	Cursor already allocated for statement
-14631	Invalid SQL statement name
-14641	Invalid SQL descriptor name
-14642	Invalid descriptor index
-14643	Invalid descriptor count
-14651	Invalid condition number
-14652	The xa_info string passed to xa_open or xa_close has an improper syntax
-14653	Invalid flags argument passed to XA routine
-14654	Asynchronous operations are not supported by XA routines
-14700	No return statement was encountered when executing a function or method
-14701	The statement is not allowed in an atomic execution context
-14702	The same column has been updated to different values when executing a trigger
-14703	An exception occurred during the execution of the trigger <%>
-14720	All possible values for the sequence <%> has been used, no more values can be generated.
-14721	There is no current value for the sequence <%> because the next value function has not been invoked
-14722	Sequence <%> locked by another user
-14723	Update of attributes for the sequence <%> failed
-14724	Not possible to allocate multiple sequence values

Number	Explanation
-14725	WITH HOLD cursors cannot be used with result set procedures
-14726	A realtime control structure (RTCS) could not be found
-14727	The databank containing the sequence is set to WORK mode, new values cannot be generated.
-14728	The operation is not allowed as the databank is temporarily set to WORK option
-14729	Table % has a UNIQUE constraint and cannot be updated while the databank option is set to WORK
-14730	Table % is involved in a FOREIGN KEY constraint and cannot be updated while the databank option is set to WORK
-14731	The primary key columns of table % cannot be updated while the databank option is set to WORK
-14732	Bind is attempted within an active transaction
-14733	The parameter value for the function builtin.uuid_from_text is invalid. The expected value is a string in the format HHHHHHHH-HHHH-HHHH-HHHH-HHHHHHHHHHHH, where each H is a hexadecimal value specified by using the symbols 0-9 and A-F.
-14734	The target area for the result of the method as_text for the distinct type builtin.uuid is too short, minimum length is 36.
-14735	When converting to coordinate, location etc. the binary length must match the underlying type
-14801	The type for an instance in a treat expression is not among the subtypes of <%>

Databank and Table Errors

These errors are associated with problems of physical access to databanks and tables. Locking errors should not result from transaction conflicts, but generally indicate either locking at the operating system level or malfunction of the internal Mimer SQL routines.

Many of the errors in this class are corrected by action taken at the operating system level. If errors persist in spite of corrective action, contact your Mimer representative.

Number	Explanation
-16001	Table <%> locked by another user
-16002	Table <%> locked by another cursor
-16003	Table <%> in referential constraint definition locked by another user
-16004	Table <%> in referential constraint definition locked by another cursor
-16005	Log locked by another user
-16006	Backup unit log file locked

Number	Explanation
-16007	Databank <%> is read only, data modifications are not allowed
-16008	Sequence <%> is located in a read only databank, new values cannot be generated
-16009	Sequence <%> is located in a databank which is offline, new values cannot be generated
-16010	The table <%> cannot be recreated since no dictionary information has been saved
-16011	The requested operation cannot be executed until previous change operation for <%> has completed
-16101	No databank <%> found in dictionary
-16135	Record no longer exists
-16141	Syntax error in filename for databank <%>
-16142	Cannot open databank <%>, file <%> not found
-16143	File protection violation for databank <%>, file <%>
-16144	Cannot open databank <%>, file <%> locked by other user
-16145	Too many databanks open concurrently (direct access I/O limit)
-16146	File create error, disk full
-16147	File create error (quota exceeded) for databank <%>, file <%>
-16148	Device or network connection not ready, databank <%>, file <%>
-16149	Cannot open databank <%> in file <%>
-16150	Tried to access databank <%> on node which is inaccessible because TRANSDB is OFFLINE
-16151	Too many databanks open concurrently
-16152	Tried to open a non-Mimer SQL databank
-16153	Data no longer available, storage device has been removed
-16154	Table control area exhausted
-16155	Incompatible version of databank <%>
-16156	Databank <%> belongs to another system databank
-16157	Tried to open a read-only databank with write access
-16159	Old version of the databank <%> cannot be accessed without using the ALTER DATABANK .. RESTORE statement
-16160	Cannot set TRANSDB shadow OFFLINE on the same node as the master TRANSDB
-16161	Databank <%> disk space exhausted

Number	Explanation
-16162	Databank LOGDB disk space exhausted
-16163	Databank TRANSDB disk space exhausted
-16164	Databank SQLDB disk space exhausted
-16172	Databank <%> locked by another Mimer SQL user
-16182	Databank <%> corrupt
-16183	Bad parameter
-16184	I/O error
-16185	Internal databank identifier invalid
-16186	Internal table identifier invalid
-16187	Shadow <%> in file <%> has illegal sequence number
-16189	Corrupt bitmap page
-16190	Table root entry not found
-16191	Exclusive access to databank required for attempted operation
-16192	Load not allowed in databank with TRANS or LOG option
-16193	TRANSDB and/or LOGDB not open
-16194	Error occurred in transaction commit phase
-16195	Internal inconsistency detected
-16196	No end of table mark found for tableid
-16197	Shadow <%> is already OFFLINE
-16198	Shadow <%> is already ONLINE
-16199	Result of bitmap page I/O undefined
-16200	Result of index page I/O undefined
-16201	Result of root page I/O undefined
-16202	Result of data page I/O undefined
-16203	Corrupt index page
-16204	Corrupt root page
-16205	Corrupt data page
-16206	Write set corrupt
-16207	Table <%> has invalid record length
-16208	Unable to open databank <%>. SHADOW license required.
-16209	Unable to open databank <%>. NETIO license required.

Number	Explanation
-16210	Cleanup control area exhausted
-16211	Not properly closed, dbcheck initiated
-16212	TRANSDB restart directory corrupt
-16213	Error when closing databank file. Please consult Mimer log file.
-16214	Blockdata DKBLK1 missing
-16215	Error accessing remote TRANSDB, node will not be accessed further
-16216	Blocksize not supported
-16217	Error when generating internal key
-16218	Operation not allowed. Configured number of users exceeded.
-16219	Too many Mimer databases started
-16220	Unable to retrieve limit on number of allowed users
-16221	Lost contact with peer
-16222	Record length from update before is invalid
-16224	Transaction state table entries exhausted
-16225	Transaction state identifier invalid
-16226	Invalid function code
-16227	Commit set corrupt
-16228	Restart set corrupt
-16229	Cancel requested
-16230	Transaction cache inconsistent
-16231	Shadow <%> is inaccessible due to incomplete CREATE SHADOW or SET ONLINE operation
-16232	Database upgrade required
-16233	Operation not allowed. Licensed number of users exceeded.
-16234	Execution interrupted by scheduler
-16235	There are pending in-doubt transactions
-16237	Unable to load collation information for tableid = <%>
-16239	Databank <%> maximum size reached, unable to insert more data
-16240	The option for databank <%> is not supported
-16241	The allocated real-time control structures have been exhausted.
-16242	An attempt was made to bind a pointer to a compressed table
-16243	Disk full will be reached for databank <%>, unable to insert more data

Number	Explanation
-16244	The page on which the real-time data value resides is currently updated elsewhere
-16245	The page checksum is invalid when reading page
-16248	Databank <%> not available, probably removed storage device
-16249	Databank LOGDB maximum size reached, unable to handle more data
-16250	Databank TRANSDB maximum size reached, unable to handle more data
-16251	Databank SQLDB maximum size reached, unable to handle more data
-16252	Invalid database pointer type
-16253	Invalid database pointer policy
-16254	Incompatible database pointer type
-16255	The result set partially overlaps a previously bound multirow database pointer
-16256	The column set does not match a previously bound multirow database pointer
-16257	Incompatible database pointer policy
-16258	Cannot lock realtime pages in memory, bufferpool region exhausted
-16259	An attempt to delete a non-existing database pointer
-16262	An attempt was made to bind a pointer to a variable format table
-16263	Bufferpool memory exhausted for blocksize <%>
-16264	One of the files for databank <%> does not match the other files in databank file set
-16265	Not all files in multi-file databank <%> are open yet
-16266	The file number for the multi-file databank <%> is incorrect
-16267	The attempted operation for the databank <%> is not supported as the databank consists of several files
-16268	Missing a file for databank <%>, please add it with ALTER DATABANK ADD FILENAME
-16269	Too many files for databank <%>, please drop extra file with ALTER DATABANK DROP FILENAME
-16270	Two files use the same file number <%> for databank <%>
-16401	Routine cannot be dropped because it is in use
-16402	Routine cannot be used because it is being altered or dropped

Miscellaneous Errors

These errors arise from miscellaneous problems that do not fall into the other classes. If the corrective action is not indicated by the error description, contact your Mimer representative for help.

Number	Explanation
-18001	Blockdata BLKDS2 not included
-18002	Cannot log in, error in SYSDB initialization
-18003	No privilege to open log file
-18004	Databank LOGDB already opened by another Mimer SQL user. Could not be opened exclusively to drop log records.
-18005	Unknown language
-18006	Language not properly installed
-18007	Unable to fetch message text
-18008	Restore in wrong sequence attempted
-18009	Mismatching version of Embedded SQL and Mimer SQL
-18010	Invalid log record found during restore operation
-18011	Mismatching version of Mimer SQL and utilities
-18012	The transformation of a TRANSDB shadow to master was interrupted before completion. Please login to BSQL to complete the transformation.
-18013	Mimer SQL started from SYSDB shadow. Transform SYSDB shadow to master with BSQL, or restart system from master SYSDB.
-18014	Alter shadow not allowed in SQL for system databanks. Use utility program instead.
-18015	Open with hold is not possible when temporary tables are used for evaluation of the query
-18016	Cursor could not be opened with hold as it is not prepared with hold
-18017	With hold functionality not supported
-18018	The network server version does not support scroll
-18019	Bad parameters passed to DBAPI4
-18020	Unknown information code = <%> used
-18021	Only SELECT, INSERT, UPDATE and DELETE operations (excluding where current forms) may be compiled together in a single statement
-18022	Distributed transactions not supported by server
-18023	Requested DTC function not supported by server
-18024	Failed to enlist transaction in distributed transaction

Number	Explanation
-18025	Unable to retrieve transaction manager's whereabouts
-18026	Failed to import transaction identifier
-18027	Statement already active in other transaction
-18028	Cannot initiate a new PSM debugger session, because the number of request threads is insufficient
-18029	Execution interrupted by debugger
-18031	A more recent version of BSQL is required to redefine system databanks
-18032	An older matching version of BSQL is required by the server to redefine system databanks
-18041	Update of primary key columns for a table located in a databank, with WORK option is not allowed
-18042	Primary key columns may not be updated by Level2 applications
-18043	The rowid column may not be updated
-18044	Inaccessible shadow found at LOAD operation
-18045	The table specified in the preceding START LOAD statement can only be referenced using single value inserts while the LOAD operation is active
-18046	WITH HOLD cursors cannot be used in XA transactions
-18047	Update or delete where current cannot modify a row that was fetched outside the current transaction
-18048	Cannot set databank option because a WITH HOLD cursor is open
-18049	Cannot set databank option because XA transaction are used
-18050	Cannot set databank option while LOAD operation is active
-18051	Cannot START LOAD because databank has temporary WORK option
-18052	Cannot use XA transactions when temporary WORK option is used
-18053	Connection rejected by server. The server security level requires the client version to be 11.0 or later.
-18054	Cannot get exclusive access to external library <%>.
-18055	Cannot open external library <%>, because it is locked
-18057	Could not set up access to external library <%>. Failed to call the initialization routine xlib_init in file <%>.
-18058	Symbol <%> not found in external library <%>, <%>
-18101	Operation not allowed. SHADOW license required
-18102	Operation not allowed. Mimer SQL license required
-18103	Operation not allowed. Mimer SQL Level2 license required

Number	Explanation
-18107	Operation not allowed. Beta test version of Mimer requires BETA license.
-18108	Cannot find a valid Mimer SQL license key
-18109	Operation not allowed. XA, distributed transaction license required
-18110	Operation not allowed, 64 bit license required
-18111	Operation not allowed, immediate restart license required
-18112	Operation not allowed, in-memory server license required.
-18121	Operation not allowed. VAR specific Mimer license required
-18122	Authorization failure. Invalid attempt to connect
-18201	SYSDB cannot be backed up using CREATE BACKUP without an ONLINE shadow
-18231	<%> records dropped from LOGDB
-18232	Shadow <%> is OFFLINE
-18233	Unable to access databank <%>, because it is OFFLINE
-18234	Error <%> occurred when trying to access shadow <%>
-18235	Error <%> occurred when trying to access databank <%>
-18236	Statistics updated for <%> tables
-18237	Databank <%> does not have LOG option
-18238	<%> records copied to incremental backup
-18239	Unable to CONNECT, because database is OFFLINE
-18240	Unable to CONNECT, because database is OFFLINE and one connection already exists
-18241	Unable to CONNECT, because SYSDB is OFFLINE
-18242	Unable to CONNECT, because SYSDB is OFFLINE and one connection already exists
-18243	Unable to access databank <%>, because database is OFFLINE
-18244	Could not get exclusive access to the database because one or more connections already exists
-18245	Could not connect to database <%>, a system administrator is executing a statement that requires exclusive access to the database
-18246	The bind-query contains multiple statements
-18247	The bind-query points to several columns
-18248	The bind-query points to several rows
-18250	The bind-query contains input parameters

Number	Explanation
-18251	The bind-query includes a scroll cursor
-18252	The bind-query is not a SELECT statement
.18253	Table Information Package mismatch
-18254	An attempt was made to bind a pointer to a primary key
-18255	An attempt was made to bind a pointer to an indexed data element
-18257	The bind-query contains a join or subselect
-18258	An update elsewhere is not yet written to stable storage
-18259	The user does not have enough privilege to perform flush
-18260	Timeout when setting CURRENT COLLATION
-18261	No critical section objects available
-18500	Database <%> not found in SQLHOSTS file
-18501	Database <%> unknown on remote node <%>
-18502	Handshake message invalid, incompatible protocol <%>
-18503	Only remote databases are allowed, specify database which is not local
-18504	The network server version of database <%> is not compatible
-18505	Local memory pool in network server exhausted (SQLPOOL)
-18506	In the current version only one local (and several remote) databases can be connected at a time
-18507	Unknown connection name <%> specified
-18508	Already connected to database <%>
-18509	Database name <%> invalid, too long or contains invalid characters
-18510	Connection name invalid, too long or contains invalid characters
-18512	Illegal reentrant call
-18513	Use another TCP/IP port number
-18514	Too deep address indirection
-18515	Cannot get value of thread-local variable
-18516	The network server version does not support Level2
-18517	MIMER_DATABASE cannot be read
-18518	Catalog version from beta test. See Release Notes, how to upgrade.
-18519	Erroneous contents in SQLHOSTS
-18520	Cannot locate the odbcini file
-18521	Error opening SQLHOSTS, filename syntax error

Number	Explanation
-18522	Error opening SQLHOSTS, file not found
-18523	Error opening SQLHOSTS, file protection violation
-18524	Error opening SQLHOSTS, file locked
-18525	Error opening SQLHOSTS, too many opened files
-18526	Error opening SQLHOSTS, file create error, disk space exhausted
-18527	Error opening SQLHOSTS, other error (-7)
-18528	Error opening SQLHOSTS, other error (-8)
-18529	Error opening SQLHOSTS, other error (-9)
-18530	Error opening SQLHOSTS, illegal access options
-18531	Client/server communication must be encrypted, but client does not support this
-18550	Invalid network package format
-18551	Unknown request code in network package (<%>)
-18552	Network package longer than expected
-18553	Internal data structures corrupt (DSNEE4)
-18554	The UTILITY program does not have client/server support
-18555	Client is using a deprecated function in the server (<%>)
-18594	Query timeout period expired
-18595	Network partner disconnected
-18601	Could not connect to database <%>, unknown node '<%>'
-18602	Could not connect to database <%>, unknown protocol '<%>'
-18603	Could not connect to database <%>, unknown interface '<%>'
-18604	Could not connect to database <%>, unknown service '<%>'
-18605	Could not connect to database <%>, chosen protocol not supported on ALPHA/VMS '<%>'
-18606	Could not connect to database <%>, network type not supported '<%>'
-18607	Could not connect to database <%>, remote node is unreachable '<%>'
-18608	Bad parameter NETID=<%> passed to network package
-18609	Invalid parameter RECLEN=<%> passed to network package
-18610	Invalid parameter BUFFER=<%> passed to network package
-18611	Too many concurrent network connections
-18612	Connection refused '<%>'

Number	Explanation
-18613	Unexpected network event '<%>'
-18614	The underlying network protocol does not have enough capabilities '<%>'
-18615	Network service busy '<%>'
-18616	Local or remote system resources are insufficient '<%>'
-18617	Connection timed out '<%>'
-18618	Insufficient privileges for attempted network operation '<%>'
-18619	Unexpected network error '<%>'
-18620	Network operation would block (asynch mode)
-18621	Could not load network library
-18622	Required routines missing from network library
-18901	Database <%> not available on node <%>
-18902	Mimer logins are currently disabled, try again later
-18903	Access denied to Mimer multi-user system
-18904	Unable to attach to multi-user system, no response
-18905	Operation not allowed. Licensed number of users exceeded
-18906	Invalid database path
-18920	Machine dependent error-18920
-18921	Machine dependent error-18921
-18922	Machine dependent error-18922
-18923	Machine dependent error-18923
-18924	Machine dependent error-18924
-18925	Machine dependent error-18925
-18926	Machine dependent error-18926
-18927	Machine dependent error-18927
-18928	Machine dependent error-18928
-18929	Machine dependent error-18929

Internal Errors

These errors arise from malfunction in internal Mimer SQL routines. Contact your Mimer representative for help.

Number	Explanation
-19001	Program level list corrupt

Number	Explanation
-19002	No program level found
-19003	Statement list corrupt
-19004	Output parameter list corrupt
-19005	Table list corrupt
-19006	Unable to find log file, LOGDB corrupt
-19007	Inconsistency detected when trying to update dictionary
-19008	Unable to open SYSTEM base tables
-19009	Dictionary table SYSTEM.USERS corrupt
-19010	Unable to extract correct information from SYSTEM.DATABANKS
-19011	Unable to extract correct information from SYSTEM.TABLE_CONSTRAINTS
-19012	Dictionary mismatch found for table with TABLE_SYSID = <%>
-19015	Sysid record in SYSTEM.OBJECTS not found
-19016	Function not supported <%>
-19017	Invalid MAE program
-19018	Invalid operation code PC=<%>
-19019	Pattern not compiled due to invalid MAE instruction sequence
-19020	Invalid function code passed to instruction <%>
-19021	No databank control block found for <%>
-19022	Bad function code passed to DSCDB3
-19023	Invalid pointer to naming structure
-19024	Severity message program corrupt
-19025	Invalid table descriptor
-19026	Invalid table descriptor, log status invalid
-19027	Base table must be opened before index tables
-19028	Table root entry not found
-19029	Unable to change position on write set because no mark is set
-19030	Invalid length for allocation of program space
-19031	Invalid table type
-19032	No table control block found
-19033	Cannot delete databank file outside transaction
-19034	Bad function code passed to DSCRD2

Number	Explanation
-19035	Invalid index descriptor
-19036	Error detected when closing table, hash chain corrupt
-19037	Invalid internal table type encountered
-19038	Write set inconsistency encountered
-19039	Invalid internal statement identifier
-19040	Invalid internal system identifier
-19041	Invalid internal user identifier
-19042	The static statement cannot be compiled because it is already identified with some other statement
-19043	The statement cannot be prepared because it is already identified with a static statement
-19044	Transaction control block chain corrupt
-19045	Shadow <%> cannot be transformed because it is OFFLINE
-19046	Databank <%> is referenced but not opened
-19047	Table has not been opened with sufficient access to allow current operation
-19048	Databank <%>, no shadow is found in dictionary with sequence number = <%>
-19049	The internal update operation has not been prepared with the old record
-19050	Catalog version not compatible with server
-19051	Compiled LIKE pattern corrupt
-19052	Could not store lookup path, inconsistency detected
-19053	Output descriptor overflow
-19054	Unable to initialize database system
-19055	Unable to generate an internal key
-19056	Inconsistent user identifier (not logged in)
-19057	Scroll program corrupt
-19058	Extended name not supported in static SQL
-19059	Invalid error message descriptor
-19061	Loss of significance for VARCHAR length
-19062	Positive overflow for VARCHAR length
-19063	Negative overflow for VARCHAR length
-19065	Bad parameter for VARCHAR length

Number	Explanation
-19066	Illegal operand for VARCHAR length
-19067	Bad record number
-19068	No matching record
-19069	Corrupt cancel state
-19071	Unrecognized data types for conversion
-19072	Invalid read set record
-19073	Insufficient internal descriptor area
-19074	Internal inconsistency encountered in TCACHE
-19075	Invalid internal DDU identifier
-19076	Internal inconsistency encountered in table list
-19077	Internal inconsistency encountered in DU1
-19078	Invalid parameter encountered in MOS
-19079	Internal inconsistency encountered in DSETH3
-19080	Thread initialization failure
-19081	Hash table missing in system control block
-19082	Internal inconsistency encountered in DSGSH2
-19083	Internal inconsistency encountered in MCOMEM
-19084	Runtime assertion failed
-19085	Invalid descriptor encountered on server
-19086	Invalid statement status encountered on server
-19087	PSM debugger resources already allocated
-19088	PSM debugger resources already deallocated
-19089	Invalid parameter encountered in DDU
-19091	Invalid function code passed to routine
-19092	Invalid function code 2 passed to routine
-19093	Invalid function code 3 passed to routine
-19094	Failed to create login response message or validation
-19095	The version of a precompiled program is incompatible with server version
-19101	Not valid conversion of data types
-19102	Not supported data type conversion
-19103	Initialization failure for data type conversion

Number	Explanation
-19111	STMs for called statements remaining after disconnect
-19112	Release of statements terminated due to error <%>
-19113	Nonexistent parameter specified
-19114	Specified collation not found
-19115	LOB identification not found in directory table
-19116	No LOB column in parameter list
-19117	LOB start position out of bounds
-19118	Client and server disagreed on LOB length
-19119	Internal truncation error encountered in Mimer client driver
-19122	Failed to load name information package for system id=<%>
-19123	Failed to load table information package for table id=<%>
-19124	Failed to load domain information package for domain id=<%>
-19125	Cannot commit transaction for WITH HOLD
-19126	WITH HOLD resource mismatch
-19127	Not allowed to use CURRENT_COLLATION
-19128	Large objects not supported by current server type
-19129	Internal inconsistency detected by SQL optimizer in <%>
-19130	Could not insert LOB value due to a LOB identification failure
-19131	Internal LOB length inconsistency
-19132	Invalid sequence of internal LOB operations
-19201	System error: <%>- Area outside MAE data storage
-19202	System error: <%>- Attempt to qqwsal() in closed area
-19203	System error: <%>- Cost value out of range
-19204	System error: <%>- Error converting into Mimer format
-19205	System error: <%>- Error from MDRCCI call
-19206	System error: <%>- Error when reading databank option
-19207	System error: <%>- Expression switch case not recognized
-19208	System error: <%>- Factor was left unused
-19209	System error: <%>- Failed to get a slave RST
-19210	System error: <%>- Generation stack underflow
-19211	System error: <%>- Group is not allocated

Number	Explanation
-19212	System error: <%>- Host variable not defined
-19213	System error: <%>- Host variable number mismatch
-19214	System error: <%>- Illegal Set Func. mode switch case
-19215	System error: <%>- Illegal Status switch case <%>
-19216	System error: <%>- Index table not found
-19217	System error: <%>- Invalid base pointer
-19218	System error: <%>- Invalid object type
-19219	System error: <%>- Invalid pointer
-19220	System error: <%>- Main switch case not recognized
-19221	System error: <%>- Multiple offset assignment
-19222	System error: <%>- Multiple restriction groups
-19223	System error: <%>- No area opened
-19224	System error: <%>- Nonexistent member
-19225	System error: <%>- No Tbl_desc for SCO
-19226	System error: <%>- NOT stack overflow
-19227	System error: <%>- NOT stack underflow
-19228	System error: <%>- Offset outside MAE data storage
-19229	System error: <%>- qqcbix() with illegal operator
-19230	System error: <%>- qqcunx() with illegal operator
-19231	System error: <%>- Error from MDRCFC call
-19232	System error: <%>- qqrlst() with NULL list
-19233	System error: <%>- qqwlst() with NULL list
-19234	System error: <%>- Query result stack underflow
-19235	System error: <%>- Query stack underflow
-19236	System error: <%>- Rule matrix index out of range
-19237	System error: <%>- Scan kind not implemented
-19238	System error: <%>- Scan stack underflow
-19239	System error: <%>- Selectivity factor value out of range
-19240	System error: <%>- Semantic stack underflow
-19241	System error: <%>- Set range violation
-19242	System error: <%>- Set size incompatibility

Number	Explanation
-19243	System error: <%>- Stack underflow
-19244	System error: <%>- Statement switch case not recognized
-19245	System error: <%>- Switch case not recognized
-19246	System error: <%>- Too complicated UNION query
-19247	System error: <%>- Too many nested subqueries
-19248	System error: <%>- Traversal stack underflow
-19249	System error: <%>- Unexpected EXPRESSION in HOST variables
-19250	System error: <%>- Unexpected expression operand
-19251	System error: <%>- Unexpected expression subtype
-19252	System error: <%>- Unexpected node class
-19253	System error: <%>- Unexpected SELECT ITEM
-19254	System error: <%>- Unexpected statement subclass
-19255	System error: <%>- Unexpected DD return code <%>
-19256	System error: <%>- Unknown Host Variable type
-19257	System error: <%>- Unknown statement type
-19258	System error: <%>- WS stack overflow
-19259	System error: <%>- X stack overflow
-19260	System error: <%>- X stack underflow
-19261	System error: <%>- Error logging is not enabled
-19262	System error: <%>- Source position line or column is negative
-19263	System error: <%>- Message insert string too long
-19264	System error: <%>- Error logging is already enabled
-19265	System error: <%>- MAE constant storage overflow
-19266	System error: <%>- Selectivity rule number out of range
-19267	System error: <%>- No entry for index id
-19268	System error: <%>- qqcfnx() with illegal operator
-19269	Unexpected duplicate row found in temporary table
-19270	System error: <%>- Scan queue underflow
-19271	System error: CPL - PSM depth overflow
-19280	System error: <%>- Error from MDRTDC call
-19290	Out of memory

Number	Explanation
-19291	Invalid attribute type
-19292	Error when trying to store procedure in dictionary
-19293	Error when trying to share program
-19300	<%> unhandled production
-19301	Internal inconsistency detected in PSM
-19302	The syntax in the view definition is not allowed when with check option is used
-19303	Recursive or cycle temporary table operation failed unexpectedly
-19310	Internal error: Invalid stream type
-19311	Internal error: Invalid stream handle
-19312	Internal error: Stream handle is in use and cannot be dropped
-19901	Function not yet implemented

Communication Errors

When an application has received a communication error, the connection will become unusable. The only valid operation on that connection will be `DISCONNECT`.

Error codes from the communication kernel layer (network routines):

Number	Explanation
-21001	Already listening on service <%>
-21002	Error trying to ASSIGN channel for TCP/IP communication
-21003	Error when creating socket
-21004	Error when binding socket address for service <%>
-21005	Error when connecting to database <%>, could not get port number for service <%>
-21006	Error when connecting to database <%>, unknown protocol <%>
-21007	Error when connecting to database <%>, unknown node <%>
-21008	Error when connecting to database <%> on <%> using <%> to service <%>
-21009	Illegal channel id specified
-21010	Error when reading data from network channel
-21011	Error when writing data to network channel
-21012	Channel is not open
-21013	Channel is not accessible from this process

Number	Explanation
-21014	Error when creating mailbox
-21015	Error when declaring network object for service <%>
-21016	Unimplemented feature
-21017	Error when accepting new channel
-21018	Error when doing local listen for database <%> on path <%>
-21019	Too many channels used
-21020	Multiple read requests on channel
-21021	Multiple write requests on channel
-21022	Local write when not owning buffer
-21023	Cancel request illegal on channel
-21024	No available channel id number
-21025	Tried to open too many local servers
-21026	Database server for database <%> not running
-21027	Incompatible buffer versions
-21028	Failed to do a LOCAL connection to the server for database <%>
-21029	Illegal reentrant request on channel
-21030	Network request would block caller
-21031	Too large network I/O requested
-21032	Could not find DSINI4 in single-user library
-21033	Could not find DSHND4 in single-user library
-21034	Could not find DSGMD4 in single-user library
-21035	Could not find DSUMP4 in single-user library
-21036	The channel is closed
-21037	The specified network interface is not supported
-21038	Could not lock communication buffers in memory
-21039	Error trying to ASSIGN channel for DECNET communication
-21040	Could not map library for single-user mode
-21041	Could not initialize CK package
-21042	Error when performing initial communication with database server
-21043	Server rejected connection to database <%> on <%> using <%> to service <%>
-21044	Server rejected named pipe connection to database <%>

Number	Explanation
-21045	The address family for network protocol was unknown
-21046	Error when creating named pipe server objects
-21047	Error when setting up TCP server objects
-21048	Unexpected communication error
-21049	Error when reading/writing data to/from network channel
-21050	Error when closing communication with database server
-21051	All local communication slots are in use
-21052	Database server request failed
-21053	Database or network service not started Error when connecting to database <%> on <%> using <%> to service <%>
-21054	Database server for database <%> not started
-21055	The Mimer network service on <%> for <%> does not currently accept new connections. Try again later.
-21056	Local communication has been disabled for database server <%>
-21057	Named pipe communication has not been enabled for database server <%>
-21058	TCP/IP communication has not been enabled for database server <%>
-21059	Library for single-user mode not self-contained
-21061	The data source name specified in connection string was not found in system information
-21062	Invalid parameters found in connection string, DSN cannot be combined with PROTOCOL, NODE, SERVICE, INTERFACE, or DIRECTORY
-21063	PROTOCOL is mandatory when any of NODE, SERVICE, INTERFACE, or DIRECTORY is specified in connection string
-21064	When PROTOCOL is specified in connection string also a DATABASE specification is required
-21065	DIRECTORY specification is required in connection string for specified PROTOCOL: <%>
-21066	NODE (host name) specification in connection string is required for specified PROTOCOL: <%>
-21067	When DRIVER is specified in connection string also a DATABASE specification is required
-21071	Remote real time connections not supported
-21072	The started server does not have real time support
-21073	Bad parameter for real time request

Number	Explanation
-21074	Error when allocating resources for real-time tasks
-21075	Internal error during flush
-21076	Cryptographic protocol failure, session is not safe and is terminated
-21077	Failed to allocate memory for cryptographic operation
-21078	Cryptographic key is exhausted, the session is terminated
-21079	Cryptographic engine is currently unavailable
-21080	A malformed network message was received
-21100	Command timed out
-21101	Error mapping MCS (Mimer Control Storage)
-21102	Error when doing system communication through the MCS
-21103	MCS communication area is busy. Try again later
-21104	Server for database <%> is already started
-21105	Illegal directory specified for the SYSDB file
-21106	Error in parameter file
-21107	Error when starting database server process
-21108	Error when looking up database name
-21109	Error when creating memory pool in database server
-21110	Could not allocate space from SQLPool
-21111	Error when initiating the ENQ/DEQ package
-21112	Error when attaching a thread to the ENQDEQ area
-21113	Error when initiating server I/O package
-21114	Error when setting default directory for database server I/O package
-21115	Could not start database server thread
-21116	Protocol error- received new request before completion of last request
-21117	Could not create proper execution environment
-21118	Database server not operational
-21119	Notification thread failed. Server can no longer respond to mimcontrol.
-21120	Illegal directory path
-21121	Could not create new directory
-21122	Channel closed by administrator
-21123	Invalid channel number specified

Number	Explanation
-21124	Error when initiating request (rq) queue
-21125	Could not lock the bufferpool in memory
-21126	Database server halted. Failed to generate automatic database dumps.
-21127	Database server halted. Dump files from the failed database are placed under <%>.
-21128	Error when stopping database server process
-21129	Error when deleting memory pool in database server
-21130	Error getting database server parameters
-21131	Must be superuser to perform this function
-21132	The environment variable MIMER_HOME must point to the Mimer distribution
-21133	An illegal combination of command switches was specified
-21134	The database parameter must be specified
-21135	Permission denied
-21136	Bad network packet length. Channel was dropped.
-21137	Failed to load mimschema_database dynamic library
-21138	Cannot locate the SYSDB databank file
-21180	Error opening SQLHOSTS file
-21181	Error opening SQLHOSTS file - file name syntax error
-21182	Error opening SQLHOSTS file - file not found
-21183	Error opening SQLHOSTS file - file protection violation
-21184	Error opening SQLHOSTS file - file is locked
-21185	Error opening SQLHOSTS file - too many files are opened
-21186	Error opening SQLHOSTS file - file creation error (diskspace exhausted)
-21187	Error opening SQLHOSTS file - machine dependent code -7
-21188	Error opening SQLHOSTS file - machine dependent code -8
-21189	Error opening SQLHOSTS file - other error
-21190	Error opening SQLHOSTS file - illegal access option
-21191	Could not find a local definition for the specified database name

Error codes used by the server when creating database dumps. These codes are never returned to application programs:

Number	Explanation
-21200	Error when creating dump file
-21201	Error when writing dump file

Error codes reflecting problems in the layer that creates and interprets network packets:

Number	Explanation
-21300	Too large network buffer requested on client side

Other file management error codes:

Number	Explanation
-21400	Illegal file name
-21401	File not found
-21402	File protection violation
-21403	File was locked
-21404	The file could not be opened since a system resource was exhausted
-21405	The disk space is exhausted
-21406	The file is not open
-21407	Read not allowed on file
-21408	Write not allowed on file
-21410	Illegal argument
-21411	Illegal character
-21412	Memory allocation error
-21499	Unspecified error

JDBC Errors

These errors arise when the Mimer JDBC Driver fails for some reason. The error codes are in the range -22000 to -22999. When using Java, the error message is always included in the exception that is thrown.

To get the complete and accurate list of error codes, execute the following command:

```
$ java com.mimer.jdbc.Driver -errors
```

Mimload Errors

The following error codes are used by the Mimload application.

Number	Explanation

-23001	Unexpected DB error
-23003	Load/Unload is not allowed within transaction
-23004	Syntax error
-23005	Out of memory
-23006	Schema not found
-23007	Databank not found
-23008	Table not found
-23009	No results were created for statement
-23010	Statement has no parameters
-23011	Output parameters are not allowed
-23012	Could not open file
-23013	Could not read from file
-23014	Could not write to file
-23015	Could not close file
-23016	Syntax error in data descriptor
-23017	Could not open log file
-23018	Could not write to log file
-23019	Could not close log file
-23020	Too long field encountered
-23021	Statement failed, see next error
-23022	Length indicator for BLOB/CLOB data invalid
-23023	Invalid escape sequence encountered

Mimer SQL C API Return Codes

The 'Definition name' in the below tables are C defines from the provided `mimerrors.h` header file.

Status Codes

Number	Definition name	Explanation
0	`MIMER_SUCCESS`	Success.
100	`MIMER_NO_DATA`	No data.

Error Codes

The Mimer API returns error codes in the range -24000 to -24999.

Number	Definition name	Explanation
-24001	MIMER_OUTOFMEMORY	An attempt to allocate heap memory failed
-24002	MIMER_SQL_NULL_VALUE	A data retrieval function returned the SQL NULL value
-24003	MIMER_TRUNCATION_ERROR	Characters were truncated when setting string or binary data
-24004	MIMER_ILLEGAL_CHARACTER	An input string contained illegal characters
-24005	MIMER_STATEMENT_CANNOT_BE_PREPARED	A statement that cannot be prepared
-24006	MIMER_UNDEFINED_COMMUNICATION	Communication feature was not defined with a call to MimerSetComRoutines
-24007	MIMER_COULD_NOT_RELEASE	Mimer API was unable to release a resource
-24010	MIMER_POSITIVE_OVERFLOW	Value was too large to fit in destination
-24011	MIMER_NEGATIVE_OVERFLOW	Value was too small to fit in destination
-24012	MIMER_UNDEFINED_FLOAT_VALUE	Floating point value was either Not-A-Number or Infinity
-24101	MIMER_SEQUENCE_ERROR	An illegal sequence of API calls was detected
-24102	MIMER_NONEXISTENT_COLUMN_PARAMETER	An API call was made referring to a column or parameter that does not exist
-24103	MIMER_UNSET_PARAMETER	Incomplete set of input parameters when executing a statement or opening a cursor

Number	Definition name	Explanation
-24104	MIMER_CAST_VIOLATION	An attempt was made to obtain column or parameter data of the wrong type
-24105	MIMER_PARAMETER_NOT_OUTPUT	An attempt was made to get the value of an input parameter
-24106	MIMER_PARAMETER_NOT_INPUT	An attempt was made to set the value of an output parameter
-24107	MIMER_PARAMETER_INVALID	A parameter to an API call was invalid
-24108	MIMER_HANDLE_INVALID	An attempt was made to call an API routine with an invalid handle
-24109	MIMER_TIMESTAMP_FORMAT_ERROR	A conversion of a TIMESTAMP from a character string failed
-24110	MIMER_ALLOCATION_FAILURE_THREAD	An attempt was made to allocate more threads than allowed
-24111	MIMER_WRONG_SERVER_TYPE	An attempt was made to establish a realtime connection to a non-realtime server
-24112	MIMER_NONEXISTENT_RECORD	The bind query does not point to a data element
-24113	MIMER_INCOMPATIBLE_POINTER_ ATTRIBUTES	The pointer type and policy are incompatible
-24114	MIMER_INVALID_POINTER_TYPE	The operation does not match the database pointer type
-24115	MIMER_UNSUPPORTED_AUTHENTICATION_ METHOD	Unsupported authentication method error
-24116	MIMER_NULL_VIOLATION	Cannot assign the null value to a non-nullable parameter
-24117	MIMER_UUID_FORMAT_ERROR	A conversion of an UUID from a character string failed

Number	Definition name	Explanation
-24414	MIMER_MEMORY_MAP_ERROR	Cannot map a shared memory
-24415	MIMER_TLS_ERROR	Error using Thread Local Storage
-24416	MIMER_INVALID_CONTROL_BLOCK	Invalid parameter
-24417	MIMER_INTERNAL_CLIENT_ERROR	An internal error in the client has occurred

Programming Dependent Errors

Number	Definition name	Explanation
-14726	MIMER_RTCS_NOT_FOUND	A realtime control structure (RTCS) could not be found.
-14732	MIMER_INVALID_TRANSACTION_STATE	Bind is attempted within an active transaction.

Databank And Table Errors

Number	Definition name	Explanation
-11015	MIMER_TASKS_EXHAUSTED	The allocated real-time tasks in the server have been exhausted.
-16241	MIMER_RTCS_EXHAUSTED	The allocated real-time control structures have been exhausted.
-16242	MIMER_TABLE_COMPRESSED	An attempt to bind a pointer to a compressed table was made. Use "ALTER TABLE nn SET COMPRESS OFF;" to resolve this.
-16244	MIMER_PAGE_UPDATED	The page on which the real-time data value resides is currently updated elsewhere.
-16252	MIMER_INVALID_RTTYPE	Invalid database pointer type.
-16253	MIMER_INVALID_RTPOLICY	Invalid database pointer policy.
-16254	MIMER_TYPE_MISMATCH	The type (single/multicol/multirow) of the database pointer is not compatible with a previously created database pointer.
-16255	MIMER_RESULT_SET_MISMATCH	The result set (data records) partially overlaps a previously bound multirow database pointer.

Number	Definition name	Explanation
-16256	MIMER_COLUMN_SET_MISMATCH	The column set does not match a previously bound multirow database pointer.
-16258	MIMER_COULD_NOT_LOCK_PAGE	Could not lock real-time pages in memory. The number of bufferpool pages in the region need to be increased.
-16259	MIMER_RTCS_INVALID	An attempt was made to delete a non-existing database pointer.
-16242	MIMER_TABLE_VARFORMAT	An attempt to bind a pointer to a variable format table was made.

Miscellaneous Errors

Number	Definition name	Explanation
-18246	MIMER_NOT_SINGLE_STATEMENT	The bind query contains multiple statements.
-18247	MIMER_NOT_SINGLE_COLUMN	The bind query points to several columns.
-18248	MIMER_NOT_SINGLE_ROW	The bind query points to several rows.
-18250	MIMER_INPUT_PARAMETER_FOUND	The bind query contains input parameters.
-18251	MIMER_SCROLL_USED	The bind query includes a scroll cursor.
-18252	MIMER_NOT_SELECT	The bind query is not a select statement.
-18253	MIMER_TIP_MISMATCH	Table Information Package mismatch.
-18254	MIMER_COLUMN_IS_PART_OF_KEY	An attempt to bind a pointer to a primary key was made.
-18255	MIMER_COLUMN_IS_PART_OF_INDEX	An attempt to bind a pointer to an indexed data element was made.
-18257	MIMER_NOT_SINGLE_TDA	The bind query contains a join or subquery.
-18258	MIMER_VOLATILE_DATA	An update elsewhere is not yet written to stable storage.

Number	Definition name	Explanation
-18259	MIMER_NO_FLUSH_PRIVILEGE	The user does not have enough privilege to perform flush.
-18261	MIMER_NO_CRITICAL_SECTION_OBJECTS	The are no critical section objects available.

Internal Errors

Number	Definition name	Explanation
-19086	MIMER_INVALID_STATEMENT_STATUS	A statement is performed in a wrong order, or using a handle from a different thread or process.
-21074	MIMER_ERROR_ALLOCATING_TASK	Error when allocating resources for real-time tasks.
-21075	MIMER_INTERNAL_FLUSH_ERROR	Internal error during flush.

MimerPy Errors

The following error codes are used by the Mimer Python interface.

Number	Explanation
-25000	Unsupported method
-25001	TPC is unsupported
-25010	Connection not open
-25011	Invalid number of parameters
-25012	Invalid number of parameters, key: <%> does not exist in dictionary
-25013	Invalid parameter format
-25014	Previous execute did not produce a result set
-25015	Cursor not open
-25016	Illegal scroll mode
-25020	Data conversion error
-25030	Out of memory
-25031	Login failure
-25101	The operation requires Mimer API version 11.0.5A or newer. You have <%>.

-25102	The operation requires Mimer API version 11.0.5B or newer. You have <%>.

Appendix C

Deprecated Features

Some non-standard features in earlier versions of Mimer SQL are deprecated, but retained for backward compatibility.

Where these features have equivalents in the standard implementation, only the standard form is documented in the main body of this manual.

Use of the standard forms is strongly recommended.

INCLUDE SQLCA

The `SQLCA` communication area is no longer supported.

Applications should now use the `SQLSTATE` or `SQLCODE` variables and the `GET DIAGNOSTICS` statement to get all the information previously obtained from `SQLCA`.

See *Communicating with the Application Program* on page *46* for a description of `SQLSTATE` and `GET DIAGNOSTICS`.

SQLDA

The SQLDA area, which was used in earlier versions of Mimer SQL, has now been replaced by a standardized SQL descriptor area. The `SQLDA` area was allocated and maintained by constructions in the host language. The SQL descriptor area is allocated and maintained by standardized ESQL statements.

Applications using SQLDA have to be modified to use SQL Descriptors instead.

VARCHAR(size) C language struct

In earlier versions of Mimer SQL a `VARCHAR` structure was documented, which was used in the C language for handling variable-length character strings.

This `VARCHAR` structure was defined as:

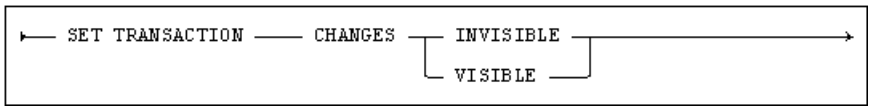
```
#define VARCHAR(x)
        struct {
            short  len;
            char   text[x];
        }
```

where the character string is stored in `text`, and the length of the text is stored in `len`.

SET TRANSACTION

Following the introduction of the `SET TRANSACTION READ` and `SET TRANSACTION ISOLATION LEVEL` options, the `SET TRANSACTION CHANGES` options no longer apply.

The following options are supported for backward compatibility in version 11.0.x only:



DBERM4

The `DBERM4` routine, which could be used to retrieve the internal Mimer SQL return code and error message text for an exception, is now deprecated.

The `GET DIAGNOSTICS` statement should now be used to retrieve these exception information items, see *Run-time Errors* on page *70*.

Index

G

H

I

J

L

M

N

O

T

U

V

W

X

Mimer SQL

User's Manual

Version 11.0

Mimer SQL, User's Manual, Version 11.0, December 2024
© Copyright Mimer Information Technology AB.

The contents of this manual may be printed in limited quantities for use at a Mimer SQL installation site. No parts of the manual may be reproduced for sale to a third party.
Information in this document is subject to change without notice. All registered names, product names and trademarks of other companies mentioned in this documentation are used for identification purposes only and are acknowledged as the property of the respective company. Companies, names and data used in examples herein are fictitious unless otherwise noted.

Produced and published by Mimer Information Technology AB, Uppsala, Sweden.

Mimer SQL Web Sites:
https://developer.mimer.com
https://www.mimer.com

Contents

Chapter 1

Introduction

Mimer SQL is an advanced database management system developed by Mimer Information Technology AB.

Mimer SQL offers a uniquely scalable and portable solution, including multi-core support. The product is available on a wide range of platforms from small embedded and handheld devices running for example Android or Linux, to workgroup and enterprise servers running Linux, Windows, macOS and OpenVMS. This makes Mimer SQL ideally suited for open environments where interoperability, portability and small footprint are important.

The database management language Mimer SQL (Structured Query Language) is compatible in all essential features with the current SQL standard, see the *Mimer SQL Reference Manual, Chapter 3, Introduction to SQL Standards*, for details.

About this Manual

This manual is intended primarily for users who have little or no experience of SQL (database query language). It provides an introduction to the concepts and usage of Mimer SQL, including how to create and how to manipulate the contents of a database.

Most examples in this manual are based on the example database, MIMER_STORE, that is described in *Appendix B The Example Environment*.

To access the example database, you can use:

- DbVisualizer, available in the Mimer SQL program group.
- Mimer BSQL, available in the Mimer SQL program group.
- Any ODBC-based SQL tool.
- Any JDBC-based SQL tool.

Mimer BSQL is a command line oriented tool for executing SQL statements, wither interactively or in scripts. This manual includes a detailed description of the facilities provided in Mimer BSQL.

Refer to the *Mimer SQL Reference Manual, Chapter 12, SQL Statements* for a complete syntax description of the SQL statements supported in Mimer SQL.

The use of Mimer SQL together with application programming interfaces such as ADO.NET, JDBC, ODBC and embedded SQL (ESQL), is described in *Mimer SQL Programmer's Manual*.

Prerequisites

There are no prerequisites for users of this manual. However, it is to the user's advantage to be familiar with the principles of the relational database model when working with SQL.

Related Mimer SQL Publications

- **Mimer SQL Reference Manual**

  Contains a complete description of the syntax and usage of all statements in Mimer SQL and is a necessary complement to this manual.

- **Mimer SQL Programmer's Manual**

  Contains a description of how Mimer SQL can be used within the context of application programs, written in conventional programming languages.

- **Mimer SQL System Management Handbook**

  Describes system administration functions, including export/import, backup/restore, databank shadowing and the statistics functionality. The information in this manual is used primarily by the system administrator, and is not required by application program developers. The SQL statements which are part of the System Management API are described in the *Mimer SQL Reference Manual*.

- **Mimer SQL platform-specific documents**

  Contain platform-specific information. A set of one or more documents is provided, where required, for each platform on which Mimer SQL is supplied.

- **Mimer SQL Release Notes**

  Contain general and platform-specific information relating to the Mimer SQL release for which they are supplied.

Suggestions for Further Reading

We can recommend the many works of C. J. Date. His insight into the potential and limitations of SQL, coupled with his pedagogical talents, makes his books invaluable sources of study material in the field of SQL theory and usage.

In particular, we can mention: A Guide to the SQL Standard (Fourth Edition, 1997). ISBN: 0-201-96426-0. This work contains much constructive criticism and discussion of the SQL standard, including SQL99.

Technical discussions on a wide range of SQL issues can be found in SQL for Smarties (Expanded 2nd edition, 1999, ISBN: 1558605762), by Joe Celko.

SQL Standards

Official documentation of the accepted SQL standards may be found in:

ISO/IEC 9075:2016(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL-2016.

Acronyms, Terms and Trademarks

IEC	International Electrotechnical Commission
ISO	International Standards Organization
SQL	Structured Query Language
PSM	Persistent Stored Modules (i.e. Stored Procedures)

All other trademarks are the property of their respective holders.

Chapter 2

Basic Concepts of Mimer SQL

This chapter provides a general introduction to the basic concepts of Mimer SQL databases and Mimer SQL objects.

Mimer SQL is a relational database system. This means that the information in the database is presented to the user in the form of tables. The tables represent a logical description of the contents of the database which is independent of, and insulates the user from, the physical storage format of the data.

The Mimer SQL database includes the data dictionary which is a set of tables describing the organization of the database and is used primarily by the database management system itself.

The database, although located on a single physical platform, may be accessed from many distinct platforms, even at remote geographical locations, linked over a network through client/server support.

Commands are available for managing the connections to different databases, so the actual database being accessed may change during the course of an SQL session.

At any one time, however, the database may be regarded as one single organized collection of information.

Tables

Data in a relational database is logically organized in tables, which consist of horizontal rows and vertical columns.

Columns are identified by a column-name. Each row in a table contains data pertaining to a specific entry in the database. Each field, defined by the intersection of a row and a column, contains a single item of data.

For example, a table containing information about currencies may have columns for the currency code, name and exchange rate:

```
CREATE TABLE currencies (
      code CHARACTER(3) PRIMARY KEY,
      currency CHARACTER(32) NOT NULL,
      exchange_rate DECIMAL(12, 4));
```

CURRENCIES

CODE	CURRENCY	EXCHANGE_RATE
AED	UAE Dirhams	3.1030
AFA	Afghanis	4092.0000
ALL	Leke	122.3000
AMD	Armenian Drams	-
ANG	Netherlands Antillian Guilders	1.4890
AOA	Kwanza	-
…	…	…

Each row in a table must have the same set of data items (one for each column in the table), but not all the items need to be filled in.

A column can have a default value defined (either as part of the column specification itself or by using a domain with a default value) and this is stored if an explicit value has not been specified.

If no default value has been defined for a column, the null value is stored when no data value is supplied (the way the null value is displayed depends on the application – in Mimer BSQL the minus sign is used).

A relational database is built up of several inter-dependent tables which can be joined together. Tables are joined by using related values that appear in one or more columns in each of the tables. Part of the flexibility of a relational database structure is the ability to add more tables to an existing database. A new table can relate to an existing database structure by having columns with data that relates to the data in columns of the existing tables. No alterations to the existing data structure are required.

All the fields in any one column contain the same data type with the same maximum length. See the *Mimer SQL Reference Manual, Chapter 6, Data Types in SQL Statements*, for a detailed description of data types supported by Mimer SQL.

Base Tables and Views

The logical representation of data in a Mimer SQL database is stored in tables (this is what the user sees, as distinct from the physical storage format which is transparent to the user).

The tables which store the data are referred to as base tables. Users can directly examine data in the base tables.

In addition, data may be presented using views, which are created from specific parts of one or more base tables or views. To the user, views may look the same as tables, but operations on views are actually performed on the underlying base tables.

Access privileges on views and their underlying base tables are completely independent of each other, so views provide a mechanism for setting up specific access to tables.

The essential difference between a table and a view is underlined by the action of the DROP command, which removes objects from the database. If a table is dropped, all data in the table is lost from the database and can only be recovered by redefining the table and re-entering the data. If a view is dropped, however, the table or tables on which the view is defined remain in the database, and no data is lost.

Data may, however, become inaccessible to a user who was allowed to access the view but who is not permitted to access the underlying base table(s).

Note: Since views are logical representations of tables, all operations requested on a view are actually performed on the underlying base table, so care must be taken when granting access privileges on views.
Such privileges may include the right to insert, update and delete information. As an example, deleting a row from a view will remove the entire row from the underlying base table and this may include table columns the user of the view had no privilege to access.

Restriction Views

Views may be created to simplify presentation of data to the user by including only some of the base table columns in the view or only by including selected rows from the base table. Views of this kind are called restriction views.

For example, a view may be created on the COUNTRIES table to include only the COUNTRY and CURRENCY_CODE columns:

```
CREATE TABLE countries (
      code CHARACTER(2) PRIMARY KEY,
      country VARCHAR(48) NOT NULL,
      currency_code CHARACTER(3) NOT NULL);

CREATE VIEW countries_view
   AS SELECT country, currency_code
   FROM countries;
```

COUNTRIES_VIEW

COUNTRY	CURRENCY_CODE
Andorra	EUR
United Arab Emirates	AED
Afghanistan	AFA
Antigua and Barbuda	XCD
Anguilla	XCD
Albania	ALL
Armenia	AMD
…	…

Similarly, a view may be created to include only the rows in COUNTRIES where US dollars are used (CURRENCY_CODE = 'USD'):

```
CREATE VIEW usd_countries_view
    AS SELECT country
    FROM countries
    WHERE currency_code = 'USD';
```

Join Views

Views may also be created to combine information from several tables – join views.

Join views can be used to present data in more natural or useful combinations than the base tables themselves provide (the optimal design of the base tables will have been governed by rules of relational database modeling).

Join views may also contain restriction conditions.

For example, the join view below presents the countries that use some kind of dollars. The `CURRENCY_CODE` in `COUNTRIES_VIEW` is linked with the `CODE` column in the `CURRENCIES` table, and a restriction of `CURRENCY` includes `'dollar'` is applied:

```
CREATE VIEW dollar_countries
   AS SELECT country, currency
   FROM countries_view JOIN currencies
      ON countries_view.currency_code = currencies.code
   WHERE lower(currency) like '%dollar%';
```

`dollar_countries`

country	currency
American Samoa	US Dollars
Anguilla	East Caribbean Dollars
Antigua and Barbuda	East Caribbean Dollars
Australia	Australian Dollars
Bahamas	Bahamian Dollars
…	…

Primary Keys and Indexes

Rows in a base table are uniquely identified by the value of the primary key defined for the table. The primary key for a table is composed of the values of one or more columns.

Primary keys are automatically indexed to facilitate effective information retrieval. The primary key index is the most effective access path for the table.

Table columns that are in the primary key, a unique constraint or used in a foreign key reference are automatically indexed (in the order in which they are defined in the key). Therefore, explicitly creating an index on these columns will not improve performance at all.

Other columns or combinations of columns may be defined as a secondary index to improve performance in data retrieval. Secondary indexes are defined on a table after it has been created (using the `CREATE INDEX` statement).

An example of when a secondary index may be useful is when a search is regularly performed on a non-keyed column in a table with many rows, defining an index on the column may speed up the search. The search result is not affected by the index but the speed of the search is optimized.

It should be noted, however, that indexes create an overhead for update, delete and insert operations because the index must also be updated.

An index will be used if the internal query optimization process determines it will improve the efficiency of a search.

An index can be used in select statements as an ordinary table, but explicit write operations on indexes are not allowed.

SQL queries are automatically optimized when they are internally prepared for execution. The optimization process determines the most effective way to execute each query, which may or may not involve using an applicable index.

WORD_SEARCH Index Algorithm

The `WORD_SEARCH` index algorithm improves performance for "begins word" searches and "match word" searches, when using the `builtin.begins_word()` and `builtin.match_word()` functions.

```
create table documents (id integer primary key, title varchar(50),
    content nvarchar(500) collate english_1);
create index dcont_ws on documents (content for word_search);
select * from documents where builtin.word_match(content, 'Mimer');
```

Data Integrity

A vital aspect of a Mimer SQL database is data integrity. Data integrity means that the data in the database is complete and consistent both at its creation and at all times during use.

Mimer SQL has built-in facilities that ensure the data integrity in the database:

- Domains
- Unique constraints and primary keys
- Foreign keys (also referred to as referential integrity)
- Check constraints in table definitions
- Check options in view definitions
- Default values
- Triggers
- Transactions

These features should be used whenever possible to protect the integrity of the database, guaranteeing that incorrect or inconsistent data is not entered into it. By applying data integrity constraints through the database management system, the responsibility of ensuring the data integrity of the database is moved from the users of the database to the database designer.

Domains

Each column in a table holds data of a single data type and length, specified when the column is created or altered. The data type may be specified explicitly (e.g. `CHARACTER(20)` or `INTEGER`) or through the use of domains.

A domain definition consists of a data type, optional check conditions and an optional default value. Data which falls outside the constraints defined by the check conditions will not be accepted in a column which is defined as belonging to the domain.

A column belonging to a domain for which a default value is defined (unless there is an explicit default value for the column) will automatically receive that value if row data is entered without a value being explicitly specified for the column.

Unique Constraints and Primary Keys

Rows in a base table are uniquely identified by the value of the primary key defined for the table. The primary key for a table is composed of the values of one or more columns. A table cannot contain two rows with the same primary key value. If the primary key contains more than one column, the key value is the combined value of all the columns in the key. Individual columns in the key may contain duplicate values as long as the whole key value is unique.

Apart from a primary key constraint its also possible to add one or more unique constraints. The primary key constraint and the unique constraint are similar, but treat NULLs in different ways. However, the definition of the primary key is also a definition of the most effective access path for the table.

Foreign Keys – Referential Integrity

A foreign key is one or more columns in a table defined as cross-referencing the primary key or a unique key of table. Data entered into the foreign key must either exist in the key that it cross-references or be null. This maintains referential integrity in the database, ensuring that a table can only contain data that already exists in the selected key of the referenced table.

As a consequence of this, a key value that is cross-referenced by a foreign key of another table must not be removed from the table to which it belongs by an update or delete operation.

The `DELETE` and `UPDATE` rules defined for the referential constraint provide a mechanism for adjusting the values in a foreign key in a way that may permit a cross-referenced key value to effectively be removed.

Note: The referential integrity constraints are effectively checked at the end of an `INSERT`, `DELETE` or `UPDATE` statement, or at `COMMIT` depending on whether the constraint is declared as IMMEDIATE or DEFERRED.

The following example illustrates the column `CURRENCY_CODE` in the table `COUNTRIES` as a foreign key referencing the primary key of the table `CURRENCIES`.

```
CREATE TABLE countries (
      code CHARACTER(2) PRIMARY KEY,
      country VARCHAR(48) NOT NULL,
      currency_code CHARACTER(3) NOT NULL,
      FOREIGN KEY (currency_code) REFERENCES currencies(code));
```

`COUNTRIES`

CODE	COUNTRY	CURRENCY_CODE
AD	Andorra	EUR
AE	United Arab Emirates	AED
AF	Afghanistan	AFA
…	…	…

CURRENCIES

CODE	CURRENCY	EXCHANGE_RATE
AED	UAE Dirhams	3.1030
AFA	Afghanis	4092.0000
ALL	Leke	122.3000
…	…	…

In this example, the referential constraint means there cannot be a currency in the COUNTRIES table that does not exist, and a currency cannot be deleted if it is assigned to a country.

Foreign key relationships are defined when a table is created using the CREATE TABLE statement and can be added to an existing table by using the ALTER TABLE statement.

The cross-referenced table must exist prior to the declaration of foreign keys on that table, unless the cross-referenced and referencing tables are the same.

The exception to this rule is when foreign key relationships are defined for tables in a CREATE SCHEMA statement. Object references in a CREATE SCHEMA statement are not verified until the end of the statement, when all the objects have been created. Therefore, it is possible to reference a table that will not be created until later in the CREATE SCHEMA statement.

Check Conditions

Check conditions may be specified in table and domain definitions to make sure that the values in a column conform to certain conditions.

Check conditions are discussed in detail in *Check Constraints* on page *102*.

Check Options in View Definitions

You can maintain view integrity by including a check option in the view definition. This causes data entered through the view to be checked against the view definition. If the data conflicts with the conditions in the view definition, it is rejected.

For example, the restriction view USD_COUNTRIES is created with the following SQL statement:

```
CREATE VIEW usd_countries
   AS SELECT code, country, currency_code
   FROM countries
   WHERE currency_code = 'USD'
   WITH CHECK OPTION;
```

This means that the view USD_COUNTRIES contains CODE, COUNTRY and CURRENCY_CODE columns from the COUNTRIES table on the condition that the value in the CURRENCY_CODE column is USD.

Any attempt to change contents of the CURRENCY_CODE column in the view or to insert data in the view where CURRENCY_CODE does not contain USD is rejected.

If check option is not used, a user could update a row in the view that is not returned by the view.

Sequences

A sequence is a database object that provides a series of integer values.

A sequence has an initial value, an increment value, a minimum value and a maximum value defined when it is created, either implicitly or explicitly (by using the `CREATE SEQUENCE` statement, see *Mimer SQL Reference Manual, Chapter 12, CREATE SEQUENCE*).

A sequence can be defined with `CYCLE` or `NO CYCLE` option. A sequence with `CYCLE` option may re-use values when the maximum value has been reached. A sequence with `NO CYCLE` option never generates the same value twice.

A sequence definition may contain a data type which determines the limits for which values that can be generated by using the sequence. The allowed data types are `SMALLINT`, `INTEGER` and `BIGINT`.

A sequence generates a series of values by starting at the initial value and proceeding in increment steps. If all values in a sequence with cycle option has been exhausted, the sequence will start over again with the min value if the increment is positive, and with the max value if the increment is negative.

It is possible to generate the next value in the value series of a sequence by using the `NEXT VALUE FOR` sequence-name construct. This is used for the first time after the sequence has been created to establish the initial value defined for the sequence. Subsequent uses will add the increment step value to the value of the sequence and the result will be established as the current value of the sequence.

It is possible to get the value of a sequence by using the `CURRENT VALUE FOR` sequence_name construct. This construct cannot be used until the initial value has been established for the sequence (i.e. using it immediately after the sequence has been created will raise an error). For each new database connection, `NEXT VALUE` must be used before `CURRENT VALUE` can be used.

When the current value of a sequence with `NO CYCLE` option is equal to the last value in the series it defines, `NEXT VALUE OF` sequence-name will raise an error and the value of the sequence will remain unaltered.

If the sequence has `CYCLE` option, `NEXT VALUE FOR` sequence-name will always succeed.

The value of `CURRENT VALUE FOR` sequence-name and `NEXT VALUE FOR` sequence-name can be used where a value-expression would normally be used. The value may also be used after the DEFAULT clause in the `CREATE DOMAIN`, `CREATE TABLE` and `ALTER TABLE` statements.

An ident must hold USAGE privilege on the sequence in order to use it.

If a sequence is dropped, with the CASCADE option in effect, all object referencing the sequence will also be dropped.

Examples:

A sequence with `CYCLE` option with start value 1, increment 3 and maximum 10 will generate the following series of values: 1, 4, 7, 10, 1, 4, 7, 10, 1, 4...

A sequence with `NO CYCLE` option, start value 1, increment 3, minvalue 1 and maxvalue 10 will generate the following series of values: 1, 4, 7, 10.

Note: It is possible that not every value in the series defined by the sequence will be generated. If a server failure occurs it is possible that some of the values in the series might be skipped.

Synonyms

A synonym is an alternative name for a table, view or another synonym. Synonyms can be created or dropped at any time.

A synonym cannot be created for a function, procedure or a module.

Using synonyms can be a convenient way to address tables that are contained in another schema.

For example, if a view called `customer_details` is contained in the schema called `mimer_store`, the full name of the view is `mimer_store.customer_details`.

This view may be referenced from the schema called `mimer_store_book` by its fully qualified name as given above.

Alternatively, a synonym may be created for the view in schema `mimer_store_book`, e.g. `cust_details`. Then the name `cust_details` can simply be used to refer to the view `mimer_store.customer_details`.

Note: The name `cust_details` is contained in schema `mimer_store_book` and can only be used in that context.

Databanks

A databank is the physical file where a tables and sequences are stored. A Mimer SQL database may include any number of databanks. There are two types of databanks - system databanks and user databanks.

System Databanks

System databanks contain system information used by the database manager. These databanks are defined when the system is created.

The system databanks are:

- `SYSDB` containing the data dictionary tables
- `TRANSDB` used for transaction handling
- `LOGDB` used for transaction logging
- `SQLDB` used in transaction handling and for temporary storage of internal work tables.

User Databanks

User databanks contain the user tables and sequences. These databanks are defined by the user(s) responsible for setting up the database. See the *Mimer SQL Reference Manual, Chapter 4, Specifying the Location of User Databanks*, for details concerning path names.

The division of tables between different user databanks is a physical file storage issue and does not affect the way the database contents are presented to the user. Except in special situations (such as when creating tables), databanks are completely invisible to the user.

Note: In Mimer SQL, backup and restore can be performed on a per-databank basis rather than on the entire database file base, see the *Mimer SQL System Management Handbook, Chapter 5, Backing-up and Restoring Data* for more information.

Shadows

Mimer SQL Shadowing can be used to create and maintain one or more copies of a databank on different disks. Shadowing provides extra protection from the consequences of disk crashes, etc.

Read more in the *Mimer SQL System Management Handbook, Chapter 10, Mimer SQL Shadowing*.

Mimer SQL Character Sets

For character data, Mimer SQL uses the character set ISO 8859-1, also known as the Latin1 character set. By default, character data is sorted in the numerical order of its code according to the `ISO8BIT` collation.

For national character data, Mimer SQL uses the Unicode character set, which is a universal character set, see https://www.unicode.org for more information. National character data is sorted according to the `UCS_BASIC` collation. `UCS_BASIC` is a collation in which the ordering is determined entirely by the Unicode scalar values of the characters in the strings being sorted.

See the *Mimer SQL Reference Manual, Appendix B, Character Sets* for more information.

Collations and Linguistic Sorting

As stated in the previous section, character and national character data is sorted according to specific collations.

A collation, also known as a collating sequence, is a database object containing a set of rules that determines how character strings are compared, searched and alphabetically sorted. The rules in the collation determine whether one character string is less than, equal to or greater than another. A collation also determines how case-sensitivity and accents are handled.

You can specify a different collation for ordering characters when you create or alter a table or create a domain.

If you have specified a collation for a column, the collation is used implicitly in SQL statements.

You only need to explicitly use a collate clause in SQL statements if you want to override the default collation or the collation you specified when creating or altering the column or creating the domain.

For more information, see *Chapter 4, Collations*.

Since Unicode is a universal character set the Unicode sorting order can be employed on any arbitrary character set that is a subset of Unicode.

The default Unicode sorting order is provided in https://www.unicode.org/reports/tr10/allkeys.txt (Unicode 3.1.1 mapping). This table (the Default Unicode Collation Element Table) provides a mapping from characters to collation elements for all the explicitly weighted characters.

Stored Procedures

In Mimer SQL you can define functions and procedures, collectively known as stored procedures.

Mimer SQL stored procedures enable you to define and use powerful functionality through the creation and execution of routines. By using stored procedures, you can move application logic from the client to the server, thereby reducing network traffic. This will also allow the logic to be shared between different applications.

Stored procedures are stored in the data dictionary and you can invoke them when needed.

For a complete and detailed discussion of stored procedures, see *Mimer SQL Reference Manual, Chapter 8, Functions* and the *Mimer SQL Programmer's Manual, Chapter 11, Mimer SQL Stored Procedures*.

Stored procedures execute their statements using the user context of the creator of the stored procedure, independent of the actual current user.

Routines – Functions and Procedures

The term routine is a collective term for functions and procedures. Functions are distinguished from procedures in that they return a single value and the parameters of a function are used for input only. A function is invoked by using it where a value expression would normally be used.

Mimer SQL supports standard procedures and also result set procedures, which are procedures capable of returning the row value(s) of a result set.

Standard procedures are invoked directly by using the `CALL` statement and can pass values back to the calling environment through the procedure parameters.

A result set procedure is invoked by using the `CALL` statement, and the result set values are presented in the same way as for a `SELECT` statement.

In Embedded SQL, ODBC and JDBC, result set procedures are invoked by declaring a cursor which includes the procedure call specification and by then using the `FETCH` statement to execute the procedure and return the row(s) of the result set.

The creator of a routine must hold the appropriate access rights on any database objects referenced from within the routine. These access rights must be held for the life of the routine.

Routine names, like those of other private objects in the database, are qualified with the name of the schema to which they belong.

Modules

A module is simply a collection of routines. All the routines in a module are created when the module is created and belong to the same schema.

If a module is dropped, all the routines contained in the module are dropped.

Under certain circumstances a routine may be dropped because of the cascade effect of dropping some other database object. If such a routine is contained in a module, it is implicitly removed from the module and dropped. The other routines contained in the module remain unaffected.

In general, care should be taken when using DROP or REVOKE in connection with routines, modules or objects referenced from within routines because the cascade effects can often affect many other objects. See *Dropping Objects from the Database* on page *113* and *Recursive Effects of Revoking Privileges* on page *123* for details.

Triggers

A trigger defines a number of procedural SQL statements that are executed whenever a specified data manipulation statement is executed on the table or view on which the trigger has been created.

The trigger can be set up to execute AFTER, BEFORE or INSTEAD OF the data manipulation statement. Trigger execution can also be made conditional on a search condition specified as part of the trigger.

Triggers are described in detail in the *Mimer SQL Programmer's Manual, Chapter 12, Triggers*.

Idents

An ident is an authorization-id used to identify users, programs and groups. There are three types of idents in a Mimer SQL database: USER, PROGRAM and GROUP idents.

USER Idents

USER idents identify individual users who can connect to a Mimer SQL database. USER idents are generally associated with specific physical individuals who are authorized to use the system.

A USER's access to the database objects is restricted by the specific privileges granted to the ident.

A USER ident is usually protected by a password. For a USER ident it is also possible to add one or several OS_USER logins which allows the user currently logged in to the operating system to access the Mimer SQL database without providing a password.

For example: if there is a USER ident ALBERT defined in Mimer SQL that has an OS_USER login ALBERT, then the operating system user ALBERT may start Mimer BSQL (for example) and connect directly to Mimer SQL simply by pressing <return> at the Username: prompt.

However, if the USER ident ALBERT defined in Mimer SQL has an OS_USER login HERBERT, then the operating system user HERBERT may start Mimer BSQL and connect directly to Mimer SQL by entering HERBERT at the Username: prompt and simply pressing <return> at the PASSWORD: prompt.

PROGRAM Idents

PROGRAM idents can be used for effective administration of access rights and authorization control.

PROGRAM idents do not strictly connect to Mimer SQL, but they may be entered by an ident by using the ENTER statement. (The ENTER statement may only be used by an ident who is already connected to a Mimer SQL database.)

An ident is granted the privilege to enter a PROGRAM ident. A PROGRAM ident is set up to have certain privileges and these apply after the ENTER statement has been used. PROGRAM idents are generally associated with specific functions within the system, rather than with physical individuals.

When a PROGRAM ident is entered, any privileges granted to that ident become current and privileges belonging to the previous ident (i.e. the ident issuing the ENTER statement) are suspended.

PROGRAM idents are disconnected with the LEAVE statement.

GROUP Idents

GROUP idents are collective identities used to define groups of user and/or program idents.

Any privileges granted to or revoked from a GROUP ident automatically apply to all members of the group. Any ident can be a member of as many groups as required, and a group can include any number of members.

GROUP idents provide a facility for organizing the privilege structure in the database system. All idents are automatically members of the group PUBLIC. When a privilege is granted to PUBLIC, all users receive the privilege.

Schemas

A schema defines a local environment within which private database objects can be created. The ident creating the schema has the right to create objects in it and to drop objects from it.

When a USER or PROGRAM ident is created, a schema with the same name usually is created and the created ident becomes the creator of the schema. This happens by default unless WITHOUT SCHEMA is specified in the CREATE IDENT statement. A user without a schema is not allowed to create any database objects at all.

When a private database object is created, the name for it can be specified in a fully qualified form which identifies the schema in which it is to be created. The names of objects must be unique within the schema to which they belong, according to the rules for the particular object-type.

If an unqualified name is specified for a private database object, a schema name equivalent to the name of the connected ident is assumed.

Access Rights and Privileges

Privileges control how users may access database objects and the operations they can perform in the database.

USER and PROGRAM idents are protected by a password, which must be given together with the correct ident name in order for a user to gain access to the database or to enter a PROGRAM ident. Passwords are stored in encrypted form in the data dictionary and cannot be read by any ident, including the system administrator. An ident's password may only be changed by the ident or by the creator of the ident.

A set of privileges define the operations each ident is permitted to perform. There are three classes of privileges in a Mimer SQL database: system, object and access.

System Privileges

System privileges, which control the right to perform backup and restore operations, the right to execute the `UPDATE STATISTICS` statement as well as the right to create new databanks, idents, schemas and to manage shadows.

System privileges are granted to the system administrator when the system is installed and may be granted by the administrator to other idents in the database. As a general rule, system privileges should be granted to a restricted group of users.

Note: An ident who is given the privilege to create new idents is also able to create new schemas.

Object Privileges

Object privileges, which control membership in `GROUP` idents, the right to invoke functions and procedures, the right to enter `PROGRAM` idents, the right to create new tables in a specified databank and the right to use a domain or sequence.

The creator of an object is automatically granted full privileges on that object.

Thus the creator of:

- a group is automatically a member of the group
- a function or procedure may execute it
- a pre-compiled statement may execute it
- a `PROGRAM` ident may enter it
- a schema may create objects in and drop objects from it
- a databank may create tables and sequences in the databank
- a table or view holds all privileges on it
- a domain may use it
- a sequence may use that sequence.

The creator of an object generally has the right to grant any of these privileges to other users, in the case of views, functions and procedures this actually depends on the creator's privileges on objects referenced from within.

Access Privileges

Access privileges, which define access to the contents of the database, i.e. the rights to retrieve data from tables or views, delete data, insert new rows, update data and to refer to table columns as foreign key references.

About Privileges

Granted privileges can be regarded as instances of grantor/privilege stored for an ident. An ident will hold more than one instance of a privilege if different grantors grant it.

A privilege will be held as long as at least one instance of that privilege is stored for the ident. All privileges may be granted with the `WITH GRANT OPTION` which means that the receiver has, in turn, the right to grant the privilege to other idents. An ident will hold a privilege with the `WITH GRANT OPTION` as long as at least one of the instances stored for the ident was granted with this option.

If the same grantor grants a privilege to an ident more than once, this will not result in more than one instance of the privilege being recorded for the ident. If a particular grantor grants a privilege without `WITH GRANT OPTION` and subsequently grants the privilege again with `WITH GRANT OPTION`, then `WITH GRANT OPTION` will be added to the existing instance of the privilege.

Each instance of a privilege held by an ident is revoked separately by the appropriate grantor. It is possible to revoke `WITH GRANT OPTION` without revoking the associated privilege completely. *Revoking Privileges* on page *121* describes revoking privileges in more detail.

The Data Dictionary

The data dictionary contains information on all the database objects (e.g. tables, views and idents) stored in a Mimer SQL database and how they relate to one another, and access rights and privileges.

The data dictionary views (INFORMATION_SCHEMA) are described in *Mimer SQL Reference Manual, Appendix 13, Data Dictionary Views*.

Mimer SQL Statements

Mimer SQL is a language made up of a number of different statements, which may be divided into the following basic categories:

- Data definition statements
- Access Control Statements
- Data manipulation statements
- Connection statements
- Transaction control statements
- Database administration statements

The SQL statements are described in detail in subsequent chapters of this manual and in the *Mimer SQL Reference Manual, Chapter 12, SQL Statements.*

In addition, there is a set of commands specific to the BSQL environment, for managing output formatting and so on, see *Chapter 9, Mimer BSQL*.

Note: In Mimer BSQL, statements are terminated by a semicolon (`;`). This is not part of the SQL statement syntax, but is included in the examples in this manual.

Data Definition Statements

Data definition statements are used to maintain objects in a database. For example:

- `CREATE`, creates objects
- `ALTER`, modifies objects

- `DROP`, drops objects
- `COMMENT`, documents objects.

Access Control Statements

Access Control Statements are used to manage privileges. For example:

- `GRANT` grants privileges
- `REVOKE` revokes privileges.

Data Manipulation Statements

Data manipulation statements are used to examine and change data in the database. For example:

- `SELECT` retrieves data
- `INSERT` adds new rows to tables
- `UPDATE` changes data in existing rows
- `DELETE` deletes data
- `CALL` executes procedures
- `SET` value assignment.

Connection Statements

Connection statements are used to connect and disconnect user and program idents to or from the database. For example:

- `CONNECT` connects a user ident to the database
- `DISCONNECT` disconnects a user ident from the database
- `SET CONNECTION` changes the active database connection
- `ENTER` enters a `PROGRAM` ident
- `LEAVE` leaves a `PROGRAM` ident.

Transaction Control Statements

Transaction control statements are used to control when database transactions begin and end, and when updates take effect. For example:

- `SET TRANSACTION` sets transaction options for subsequent transactions
- `SET SESSION` sets the default transaction options for the session
- `START` starts a transaction build-up
- `COMMIT` commits the current transaction
- `ROLLBACK` abandons the current transaction.

Database Administration Statements

Database administration statements are used to manage backup/restore operations and the statistical information used to optimize queries. For example:

- `CREATE BACKUP` creates a backup copy of a databank, with an optional incremental backup. Incremental backups may also be taken on their own with the statement `CREATE INCREMENTAL BACKUP`
- `ALTER DATABANK`, the `RESTORE` variant of this statement recovers a databank from incremental backup information
- `SET DATABASE` sets the database `ONLINE` or `OFFLINE`
- `SET DATABANK` sets a databank `ONLINE` or `OFFLINE`
- `SET SHADOW` sets one or more shadows `ONLINE` or `OFFLINE`
- `UPDATE STATISTICS` updates the statistical information used for query optimization. `DELETE STATISTICS` deletes the statistical information.

Chapter 3

Retrieving Data

This chapter describes how to retrieve information from a Mimer SQL database. In a relational database, information retrieved from one or more source tables is returned in the form of a result table, also called a result set.

The statement used to retrieve information is `SELECT`.

The examples in this chapter are based on the example database included with the Mimer SQL distribution, see *Appendix B The Example Environment*.

Simple Retrieval

The simplest retrievals fetch information from one table.

The general form of the simple `SELECT` statement is:

```
SELECT column-list FROM table_name WHERE condition;
```

The column-list specifies which columns to select, and the `WHERE` condition determines which rows to select. If no `WHERE` condition is specified, all rows are retrieved from the source table or view.

Examples of Simple Retrieval

Find the format identifiers and their meaning:

```
SELECT format_id, format
FROM formats;
```

Returns:

FORMAT_ID	FORMAT
1	Audio CD
2	Cassette
3	DVD Audio
4	Vinyl
5	Audio Cassette
6	Audio CD
7	Hardcover
8	Paperback

FORMAT_ID	FORMAT
9	DVD Video
10	Video

Find the name and code for all countries that use Australian dollars (AUD).

```
SELECT country, code
FROM countries
WHERE currency_code = 'AUD';
```

Returns:

COUNTRY	CODE
Australia	AU
Cocos (Keeling) Islands	CC
Christmas Island	CX
Heard and McDonald Islands	HM
Kiribati	KI
Norfolk Island	NF
Nauru	NR
Tuvalu	TV

How to formulate selection conditions is described in detail in *Selecting Specific Rows* on page *27*.

Result Order

The columns in the result table are ordered as they are written in the `SELECT` statement, irrespective of the ordering in the source table. For example:

```
SELECT format, format_id
FROM formats;
```

Returns:

FORMAT	FORMAT_ID
Audio CD	1
Cassette	2
DVD Audio	3
Vinyl	4
Audio Cassette	5
Audio CD	6
Hardcover	7
Paperback	8
DVD Video	9
Video	10

Quick Select

A shorthand form for selecting all columns from a table is:

```
SELECT * FROM table_name ...
```

In this case, the columns in the result table are ordered as they are defined in the source table.

Table Names

A table name in a `SELECT` statement may be qualified by the name of the schema to which the table belongs in the form `schema.table`.

Unqualified table names are implicitly qualified by the ident name of the current user.

The table name must be written in its qualified form if the name of the schema to which the table belongs differs from the name of the current user.

For example:

```
SELECT *
FROM mimer_store.formats;
```

Returns:

FORMAT_ID	FORMAT	CATEGORY_ID	DISPLAY_ORDER
1	Audio CD	1	20
2	Cassette	1	30
3	DVD Audio	1	10
4	Vinyl	1	40
5	Audio Cassette	2	40
6	Audio CD	2	30
7	Hardcover	2	10
8	Paperback	2	20
9	DVD Video	3	10
10	Video	3	20

Setting Column Labels

Columns in the result table normally have the same name as the corresponding columns in the source table.

By using an `AS` clause after the column name in the `SELECT` statement, the column in the result table can be given an alternative name.

`AS` clauses can be used for as many columns as required. A label may be up to 128 characters long, and follows the same syntax rules as column names, see the *Mimer SQL Reference Manual, Chapter 6, SQL Identifiers*.

For example:

```
SELECT format AS format_name, category_id
FROM formats;
```

Returns:

FORMAT_NAME	CATEGORY_ID
Audio CD	1
Cassette	1
DVD Audio	1
Vinyl	1
Audio Cassette	2
Audio CD	2
Hardcover	2
Paperback	2
DVD Video	3
Video	3

Labels are particularly useful in queries that retrieve computed values, where the result table column is otherwise unnamed, see *Retrieving Computed Values* on page *34*.

Eliminating Duplicate Values

The simple `SELECT` statement retrieves all rows which fulfill the selection conditions. The result may contain duplicate values.

For example:

```
SELECT category_id
FROM formats;
```

Returns:

CATEGORY_ID
1
1
1
1
2
2
2
2
3
3

By adding the keyword DISTINCT before the column list you can eliminate all duplicate rows from the result table.

The keyword DISTINCT may only be used once in a simple SELECT statement.

For example:

```
SELECT DISTINCT category_id
FROM formats;
```

Returns:

CATEGORY_ID
1
2
3

DISTINCT also eliminates duplicate rows containing null values, although technically null is not regarded as equal to null, see *Handling Null Values* on page *67*.

If the selected columns include the whole primary key in the source table, the keyword DISTINCT is unnecessary, since all rows in the result table will be unique. Remember however that a view may contain duplicate rows, so that selecting all columns does not always guarantee that the result does not contain duplicate rows.

Selecting Specific Rows

Rows are selected in the SELECT statement according to the search condition in the WHERE clause. This condition relates column value(s) to expressions.

Comparison Conditions and WHERE

Comparison operators that may be used in the WHERE clause are:

Operator	Explanation
=	equal to
<>	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

Comparisons can be combined in the search condition using the logical operators AND and OR, and reversed using NOT.

Each comparison must be expressed in full; for example

```
WHERE price >= 10.00 AND price <= 20.00
```

may not be expressed as

```
WHERE price >= 10.00 AND <= 20.00
```

Comparing Character Strings

Character strings are compared character by character from left to right.

If strings are of different lengths, the shorter is conceptually padded to the right with blanks before the comparison is made (i.e. character difference takes precedence over length difference).

The default collation for characters is an extended Latin1 character set as defined by ISO 8859-1, see the *Mimer SQL Reference Manual, Appendix B, Character Sets* for the exact sequence. Default collation for Unicode characters (national character data) is the UCS_BASIC collation.

For more information on collations, see the *Mimer SQL User's Manual, Chapter 4, Collations*.

Retrieve the European Article Number (EAN), price and number in stock for all available items costing 100 euros and more:

```
SELECT ean_code, price, stock
FROM items
WHERE status = 'A'
  AND price >= 100.00;
```

Returns:

EAN_CODE	PRICE	STOCK
790051157548	115.98	14
790051155506	279.98	16
790051595920	227.98	10

Comparing Temporal Data

When stating conditions on temporal data in tables, datetime and interval literals can be specified. There are also the `CURRENT_DATE`, `LOCALTIME`, `LOCALTIMESTAMP` and `BUILTIN.UTC_TIMESTAMP` functionality which read the server clock and return that value.

If there is more than one occurrence of these pseudo literals in a statement the clock is only read once.

List the EAN and price for any items released on September 5, 1994:

```
SELECT ean_code, price
FROM items
WHERE release_date = DATE'1994-09-05';
```

Returns:

EAN_CODE	PRICE
9780001006041	7.00

Retrieve the EAN and price for any items with a release date in the future:

```
SELECT ean_code, price
FROM items
WHERE release_date > CURRENT_DATE;
```

Returns:

EAN_CODE	PRICE
7298976754871	13.98
7464376662256	15.98
9781990789861	13.99
9781993789639	6.99

For an example of interval literals, see *Datetime Arithmetic and Functions* on page *42*.

Pattern Conditions

`LIKE` is used to search for character strings that match a specified pattern.

Patterns in the `LIKE` condition are written with wildcard characters (also called meta-characters):

Pattern	Explanation
_	(underscore) stands for any single character
%	stands for any sequence of zero or more characters

(Wildcards only have significance in `LIKE` predicates.)

Find all currencies whose names include the string "Islands":

```
SELECT currency
FROM currencies
WHERE currency LIKE '%Islands%';
```

Returns:

CURRENCY
Falkland Islands Pounds
Cayman Islands Dollars
Solomon Islands Dollars

Find all formats whose names do not contain the string “Audio”:

```
SELECT format
FROM formats
WHERE format NOT LIKE '%Audio%';
```

Returns:

FORMAT
Cassette
Vinyl
Hardcover
Paperback
DVD Video
Video

Remember that character strings in SQL statements are always written within single quotation marks (' ').

A `LIKE` predicate where the pattern string does not contain any wildcard characters is essentially equivalent to a basic predicate using the `'='` operator, except that comparison strings in an `'='` comparison are conceptually padded with blanks whereas those in the `LIKE` comparison are not.

For example:

```
'Dollars ' =    'Dollars'  is true
'Dollars ' LIKE 'Dollars ' is true
'Dollars ' LIKE 'Dollars%' is true
```

but

```
'Dollars ' LIKE 'Dollars' is false
```

The `LIKE` predicate may include an `ESCAPE` clause defining a character which is used to ‘escape’ wildcard characters. A wildcard character immediately following an escape character is taken at face value. See the *Mimer SQL Reference Manual, Chapter 9, The LIKE Predicate*, for more details.

More about Searching for Character Strings

Some other examples of searching for character strings are:

`LIKE '%P%'`	matches any string that contains an upper-case ‘P’.
`LIKE '%P%' COLLATE english_1`	matches any string that contains an upper or lower case ‘P’.
`LIKE '_abc'`	matches any four letter character string ending with lower-case ‘abc’.
`LIKE '%A\%' ESCAPE '\'`	matches any string ending with ‘A%’.
`LIKE 'D_d_'`	matches any four letter string with D and d in the first and third positions respectively. Examples of possible values: Dude, Dads.

Set Conditions

IN and NOT IN

The operator `IN` finds the values that are contained in a set of values. The set is given as a comma-separated list enclosed in parentheses.

`NOT IN` finds values which are not contained in the specified set.

Which currencies are represented by the codes "SEK" or "GBP"?

```
SELECT currency
FROM currencies
WHERE code IN ('SEK', 'GBP');
```

Returns:

CURRENCY
Pounds Sterling
Swedish Kronor

List all the formats other than those for identifiers 1, 5 and 7:

```
SELECT format, format_id
FROM formats
WHERE format_id NOT IN (1, 5, 7);
```

Returns:

FORMAT	FORMAT_ID
Cassette	2
DVD Audio	3
Vinyl	4
Audio CD	6
Paperback	8
DVD Video	9
Video	10

Note: `NOT IN` is undefined if the subquery's result contains a null value. E.g. `SELECT * FROM tab WHERE 1 NOT IN (3, <null>, 4)` will return an empty result set.

BETWEEN and NOT BETWEEN

The operators `BETWEEN` and `NOT BETWEEN` are used to find values that are within or outside an interval. The interval includes the limits specified in the `BETWEEN` condition.

Find the EAN and release date for EANs outside the "Bookland" range (e.g. 978 prefix) that were released during January 1998:

```
SELECT ean_code, release_date
FROM items
WHERE ean_code NOT BETWEEN 9780000000000 AND 9789999999999
  AND release_date BETWEEN DATE'1998-01-01' AND DATE'1998-01-31';
```

Returns:

EAN_CODE	RELEASE_DATE
90431587720	1998-01-05
93624662426	1998-01-13
45778040629	1998-01-20

`BETWEEN` may also be used for character comparisons.

For example:

```
SELECT code, country
FROM countries
WHERE country BETWEEN 'South Africa' AND 'Suriname';
```

Returns:

CODE	CURRENCY
ES	Spain
LK	Sri Lanka
SD	Sudan
SR	Suriname
ZA	South Africa

Find which currencies have an exchange rate in the range of 1 to 2:

```
SELECT currency
FROM currencies
WHERE exchange_rate BETWEEN 1.00 AND 2.00;
```

Returns:

CURRENCY
Netherlands Antillian Guilders
Australian Dollars
Aruban Guilders
Convertible Marka
Barbados Dollars
Leva
…

BETWEEN SYMMETRIC

`BETWEEN` has the `SYMMETRIC` option which is used to verify the interval's lower and upper limits in both directions. This is especially useful when writing queries where the `BETWEEN` limit values are not defined until run-time, or where the limits are column or function references.

BETWEEN SYMMETRIC example with host variables:

```
SELECT code, country
FROM countries
WHERE country BETWEEN SYMMETRIC :country1 AND :country2;
country1: Suriname
country2: South Africa
```

Returns:

CODE	CURRENCY
ES	Spain
LK	Sri Lanka
SD	Sudan
SR	Suriname
ZA	South Africa

Without `SYMMETRIC` specified, this query would have returned an empty result set.

BETWEEN SYMMETRIC example with columns as arguments:

```
SELECT *
FROM table1 JOIN table2
ON table1.col1 BETWEEN SYMMETRIC table2.col1 AND table2.col2;
```

Retrieving Computed Values

You can retrieve computed values by using arithmetic, string and boolean operators in the `SELECT` clause of the statement.

When retrieving computed values, parentheses can be used to force the operation priority.

Without parentheses, the normal precedence rules apply. See *Mimer SQL Reference Manual, Chapter 7, Operator Precedence*, for information regarding operator precedence.

The following computational operators may be used:

Operator	Explanation
+	addition
-	subtraction
*	multiplication
/	division
||	string concatenation

See the *Mimer SQL Reference Manual, Chapter 6, SQL Syntax Elements*, for information regarding the type and precision of the result of an arithmetic expression.

Show the exchange rate for the US Dollar if there was a -10% change:

```
SELECT exchange_rate, exchange_rate * 0.90
FROM  currencies
WHERE code = 'USD';
```

Returns:

EXCHANGE_RATE	
0.8711	0.783990

Evaluating Boolean Expressions

Boolean expressions return a truth value (`TRUE` or `FALSE`), depending on the result of one or more boolean expressions.

Specify when the exchange rate is less than 1 or the currency code is “ALL”:

```
SELECT currency,
       exchange_rate < 1.0 or code = 'ALL'
FROM currencies;
```

Returns:

currency	
UAE Dirhams	FALSE
Afghanis	FALSE
Leke	TRUE

currency	
Armenian Drams	-
Netherlands Antillian Guilders	FALSE
Kwanza	-
Argentine Pesos	TRUE
Australian Dollars	FALSE
…	…

Labels and Computed Values

The computed column is unnamed by default in the result table. A label may be used to provide a name.

For example:

```
SELECT exchange_rate, exchange_rate * 0.90 AS new_exchange_rate
FROM currencies
WHERE code = 'USD';
```

Returns:

EXCHANGE_RATE	NEW_EXCHANGE_RATE
0.8711	0.783990

Constant Values

A column may also be computed as a constant value, which adds an extra column to the result table.

For example:

```
SELECT exchange_rate, '10% reduction:',
       exchange_rate * 0.90 AS new_exchange_rate
FROM currencies
WHERE code = 'USD';
```

Returns:

EXCHANGE_RATE		NEW_EXCHANGE_RATE
0.8711	10% reduction:	0.783990

You may also retrieve a value computed using the values in two or more columns, providing that the data types are compatible.

Retrieve the currencies prefixed with the word "Currency:":

```
SELECT 'Currency: ' || currency
FROM currencies
WHERE code LIKE 'A%';
```

Returns:

Currency: UAE Dirhams
Currency: Afghanis
Currency: Leke
Currency: Armenian Drams
Currency: Netherlands Antillian Guilders
Currency: Kwanza
…

Padding Concatenated Strings

For string concatenation, column values are padded with trailing blanks to the length of the column definition, if the column data type is fixed-length (`CHARACTER` or `NATIONAL CHARACTER`).

For example:

```
SELECT currency || 'Currency'
FROM currencies
WHERE code LIKE 'A%';
```

Returns:

UAE Dirhams	Currency
Afghanis	Currency
Leke	Currency
Armenian Drams	Currency
Netherlands Antillian Guilders	Currency
Kwanza	Currency
…	

(If the column data type is variable length, i.e. `VARCHAR` or `NCHAR VARYING`, no blank padding is performed.)

Using Scalar Functions

Scalar functions operate on expressions or on a single value received from a SELECT statement.

Some of the scalar functions available are:

Scalar function	Description
CHAR_LENGTH	returns the length of a string.
EXTRACT	returns a single field from a DATETIME or INTERVAL value.
LOWER	converts all upper case letters in a character string to lower case.
POSITION	returns the starting position of the first occurrence of a specified string expression, starting from the left, in the given character string.
SOUNDEX	returns a character string containing six digits which represents an encoding of the sound of the given character string.
SUBSTRING	extracts a substring from a given string, according to specified start position and length of the substring.
TRIM	removes leading and/or trailing instances of a specified character from a string.
UPPER	converts all lower case letters in a character string to upper case.

The complete list of scalar functions can be found in the *Mimer SQL Reference Manual, Chapter 8, Scalar Functions*.

Examples of Scalar Functions

The following are examples that illustrate how the scalar functions may be used:

List all currencies that contain the letters "AU" in upper or lower case:

```
SELECT currency
FROM currencies
WHERE LOWER(currency) LIKE '%au%';
```

Returns:

CURRENCY
Australian Dollars
Mauritius Rupees
Saudi Riyals

Note: Alternatively, a case insensitive collation can be used to get the same result.

```
SELECT currency
FROM currencies
WHERE currency COLLATE english_1 LIKE '%au%';
```

Find the position of the first space character in the formats column:

```
SELECT format, POSITION(' ' IN format)
FROM formats;
```

Returns:

FORMAT	
Audio CD	6
Cassette	0
DVD Audio	4
Vinyl	0
Audio Cassette	6
Audio CD	6
Hardcover	0
Paperback	0
DVD Video	4
Video	0

Append the word “Currency” to the currencies (without trailing spaces):

```
SELECT TRIM(TRAILING FROM currency) || ' Currency'
FROM currencies
WHERE code LIKE 'A%';
```

Returns:

UAE Dirhams Currency
Afghanis Currency
Leke Currency
Armenian Drams Currency
Netherlands Antillian Guilders Currency
Kwanza Currency
…

Remove both leading and trailing spaces from the currencies and convert to upper-case; and get the significant length (in characters):

```
SELECT UPPER(TRIM(currency)), CHAR_LENGTH(TRIM(currency))
FROM currencies;
```

Returns:

UAE DIRHAMS	11
AFGHANIS	8

LEKE	4
ARMENIAN DRAMS	14
NETHERLANDS ANTILLIAN GUILDERS	30
KWANZA	6
...	...

Find the country that sounds the same as "ASTRALYA":

```
SELECT country
FROM countries
WHERE SOUNDEX(country) = SOUNDEX('astralya');
```

Returns:

COUNTRY
Australia

Extract the first 5 characters from each format:

```
SELECT SUBSTRING(format FROM 1 FOR 5)
FROM formats;
```

Returns:

FORMAT
Audio
Casse
DVD A
Vinyl
Audio
Audio
Hardc
Paper
DVD V
Video

Using the CASE Expression

With a `CASE` expression, it is possible to specify a conditional value. Depending on the result of one or more conditional expressions, the `CASE` expression can return different values.

The rules for `CASE` expressions are fully described in the *Mimer SQL Reference Manual, Chapter 9, CASE Expression*.

Case Expression Examples

The following select statements presents two examples of how CASE expressions can be used.

Simple Case Expression

Give a textual description to the DISPLAY_ORDER column and display them in the numeric order:

```
SELECT category_id,
       CASE display_order
          WHEN 10 THEN 'FIRST'
          WHEN 20 THEN 'SECOND'
          WHEN 30 THEN 'THIRD'
          WHEN 40 THEN 'FOURTH'
          ELSE 'UNKNOWN'
       END,
       format
FROM formats
ORDER BY category_id, display_order, format;
```

Returns:

CATEGORY_ID		FORMAT
1	FIRST	DVD Audio
1	SECOND	Audio CD
1	THIRD	Cassette
1	FOURTH	Vinyl
2	FIRST	Hardcover
2	SECOND	Paperback
2	THIRD	Audio CD
2	FOURTH	Audio Cassette
3	FIRST	DVD Video
3	SECOND	Video

This form of a case expression is known as a simple case expression, in which an operand is compared to a list of values.

If there is a match in one of the when clauses, the result is the value to the right of the then clause.

If none of these matches, the value in the else clause is returned.

If there is no else clause in a case expression and no when clause matches, a null value is returned.

Case Expression

The other form of the case expression can be seen in the following example.

Display the word "UNKNOWN" if the EXCHANGE_RATE value is undefined (i.e. null); and display the word "PARITY" if the rate is 1 to 1; otherwise do not display anything:

```
SELECT currency,
       CASE
          WHEN exchange_rate IS NULL THEN 'UNKNOWN'
          WHEN exchange_rate = 1.0 THEN 'PARITY'
          ELSE ''
       END
FROM currencies
WHERE code LIKE 'A%';
```

Returns:

CURRENCY	
UAE Dirhams	
Afghanis	
Leke	
Armenian Drams	UNKNOWN
Netherlands Antillian Guilders	
Kwanza	UNKNOWN
…	…

In this form it is possible that more than one of the when clauses evaluates to true, in which case the value in the first (from left) of the matching clauses is returned.

Using the CAST Specification

The `CAST` specification explicitly converts data of one data type to another data type.

Conversion between data types is allowed if the rules for assignment to the target data type are not violated. See the *Mimer SQL Reference Manual, Chapter 9, CAST Specification* for more information.

Show the exchange rate for the US Dollar with a -10% change, force the result to four decimal places:

```
SELECT CAST(exchange_rate * 0.90 AS DECIMAL(12, 4))
FROM currencies
WHERE code = 'USD';
```

Returns:

0.7839

Retrieve the EAN, price and release date for any items released in 1987. Convert the release date (`DATE'YYYY-MM-DD'`) to character with the format MM/DD/YY:

```
SELECT ean_code, price,
       SUBSTRING(CAST(release_date AS CHAR(26)) FROM 11 FOR 2)
       || '/' ||
       SUBSTRING(CAST(release_date AS CHAR(26)) FROM 14 FOR 2)
       || '/' ||
       SUBSTRING(CAST(release_date AS CHAR(26)) FROM 8 FOR 2) AS date
FROM items
WHERE EXTRACT(YEAR FROM release_date) = 1987;
```

Returns:

EAN_CODE	PRICE	DATE
9780006167242	4.99	04/30/87
9780002315432	15.99	06/15/87

Datetime Arithmetic and Functions

It is possible to use datetime and interval values in expressions to calculate new datetime and interval values.

Valid operations are:

- addition or subtraction between an interval value and a datetime value
- subtracting a datetime from another datetime value
- adding or subtracting two interval values
- multiplying or dividing an interval by a numerical value

The first of these operations yields a datetime value while the others result in an interval value.

Retrieve the EAN, price and the number of days to the release date for any items with a release date in the future:

```
SELECT ean_code, price,
       (release_date - CURRENT_DATE) DAY(3) AS days
FROM items
WHERE release_date > CURRENT_DATE;
```

Returns:

EAN_CODE	PRICE	DAYS
7298976754871	13.98	5
7464376662256	15.98	12
9781990789861	13.99	8
9781993789639	6.99	4

When taking the difference between two datetime values it is necessary to specify the type of the resulting interval.

It is also possible to specify the precision of the interval as shown in the above example; the default precision for day is 2.

Retrieve the EAN, price and the release date for any items with a release date in the next 100 hours:

```
SELECT ean_code, price, release_date
FROM items
WHERE CAST(release_date AS TIMESTAMP)
         BETWEEN LOCALTIMESTAMP
         AND LOCALTIMESTAMP + INTERVAL '100' HOUR(3);
```

Returns:

EAN_CODE	PRICE	RELEASE_DATE
9781993789639	6.99	2002-03-15

About Intervals

SQL distinguishes between `YEAR-MONTH` (long) intervals and `DAY-TIME` (short) intervals.

`YEAR-MONTH` intervals are: `YEAR`, `MONTH` and `YEAR TO MONTH`.

`DAY-TIME` intervals are: `DAY`, `HOUR`, `MINUTE`, `SECOND`, `HOUR TO MINUTE`, `HOUR TO SECOND`, `MINUTE TO SECOND`, `DAY TO HOUR`, `DAY TO MINUTE` and `DAY TO SECOND`.

Long intervals may only be compared to other long intervals, and short intervals may only be compared to other short intervals, i.e. short and long intervals are not comparable.

Extracting Values

It is possible to extract part of a datetime value with the EXTRACT function. The function returns a numeric value.

Extract the month and year for any items with a release date in the next 4 days:

```
SELECT CASE EXTRACT(MONTH FROM release_date)
          WHEN  1 THEN 'January'
          WHEN  2 THEN 'February'
          WHEN  3 THEN 'March'
          WHEN  4 THEN 'April'
          WHEN  5 THEN 'May'
          WHEN  6 THEN 'June'
          WHEN  7 THEN 'July'
          WHEN  8 THEN 'August'
          WHEN  9 THEN 'September'
          WHEN 10 THEN 'October'
          WHEN 11 THEN 'November'
          WHEN 12 THEN 'December'
       END
       || ' ' ||
       CAST(EXTRACT(YEAR FROM release_date) AS CHAR(4))
FROM items
WHERE release_date BETWEEN CURRENT_DATE
                       AND CURRENT_DATE + INTERVAL '4' DAY;
```

Returns:

March 2002

DAYOFWEEK

Another useful function is DAYOFWEEK which returns the day number within a week. Monday has the value 1 and Sunday has the value 7.

Find the release day for any items with a release date in the next 4 days:

```
SELECT CASE DAYOFWEEK(release_date)
          WHEN 1 THEN 'Monday'
          WHEN 2 THEN 'Tuesday'
          WHEN 3 THEN 'Wednesday'
          WHEN 4 THEN 'Thursday'
          WHEN 5 THEN 'Friday'
          WHEN 6 THEN 'Saturday'
          WHEN 7 THEN 'Sunday'
       END
FROM items
WHERE release_date BETWEEN CURRENT_DATE
                       AND CURRENT_DATE + INTERVAL '4' DAY;
```

Result:

Friday

Using Set Functions

The functions listed below can be used in the column list of the SELECT statement to retrieve the result of the function on a specified column.

Set function	Explanation
AVG	average of values (numerical columns only)
COUNT	number of rows
MAX	largest value
MIN	smallest value
SUM	sum of values (numerical columns only)

About Set Functions

Set functions in SELECT statements are applied to data in the result table, not in the source table.

Set functions return a single value for the whole table unless a GROUP BY clause is specified, see *Grouped Set Functions – the GROUP BY Clause* on page *47*.

For all set functions, null values are eliminated from the column before the function is applied. The special form COUNT(*) counts the number of rows including null values.

The keywords ALL and DISTINCT may be used to qualify set functions. ALL gives a result based on all values including duplicates. DISTINCT eliminates duplicates before applying the function. If neither keyword is specified, duplicates are not removed.

Note: Set functions may not be used together with direct column references in the SELECT list (unless the SELECT statement includes a GROUP BY clause, see *Grouped Set Functions – the GROUP BY Clause* on page *47*).

Example of Set Functions

The set functions are illustrated with results from the following table:

Note: A hyphen - indicates null.

SAMPLE
1.0
2.0
2.0
2.0
3.0
3.0
4.0
5.0
-
-

```
COUNT(SAMPLE)8
COUNT(*)10
COUNT(DISTINCT SAMPLE)5
SUM(SAMPLE)22.0
SUM(ALL SAMPLE)22.0
SUM(DISTINCT SAMPLE)15.0
AVG(SAMPLE)2.75000000000
AVG(ALL SAMPLE)2.75000000000
AVG(DISTINCT SAMPLE)3.00000000000
MAX(SAMPLE)5.0
MIN(SAMPLE)1.0
```

Note: `AVG(column)` is equivalent to `SUM(column)/COUNT(column)`. However, the expression `SUM(column)/COUNT(*)` will give a different answer if the column includes null values.

Thus, for the table above:

```
SUM(SAMPLE)/COUNT(SAMPLE)     2.75000000000   (22/8)
SUM(SAMPLE)/COUNT(*)          2.20000000000   (22/10)
```

More Set Functions Examples

Some further examples of set functions applied to the example database are given below.

How many rows are there in the CURRENCIES table?

```
SELECT COUNT(*)
FROM currencies;
```

How many currencies have a defined exchange rate (i.e. EXCHANGE_RATE is not null)?

```
SELECT COUNT(exchange_rate)
FROM currencies;
```

What is the average exchange rate?

```
SELECT AVG(exchange_rate)
FROM currencies;
```

Decimal Calculation

The AVG function returns an integer if the operand is an integer, and a decimal if the operand is decimal. To force decimal calculation of averages from an integer column, cast the column operand as decimal:

```
SELECT AVG(CAST(column AS DECIMAL)) ...
```

Grouped Set Functions – the GROUP BY Clause

Normally, set functions return a single value, calculated from the set of all values in the column or expression.

If the SELECT statement includes a GROUP BY clause, set functions will be applied to groups of values. Columns used for GROUP BY do not have to be included in the SELECT list.

Find the number of rows in each category within the FORMATS table:

```
SELECT category_id, COUNT(*)
FROM formats
GROUP BY category_id;
```

Returns:

CATEGORY_ID	
1	4
2	4
3	2

Restrictions When Using GROUP BY

Using a GROUP BY clause places some restrictions on the SELECT statement.

Only constants, columns listed in the GROUP BY clause, and columns used as arguments to set functions may be included in the SELECT list.

A derived table can be used to overcome this restriction.

Find the number of released items, grouped by year, month and format:

```
select y, m, format, count(*)
from
(
    select extract(year from release_date) as y,
           extract(month from release_date) as m,
           format
    from product_details
) dt
group by y, m, format;
```

Null Values

For grouping purposes, null values are regarded as equivalent. Thus for the example table:

SAMPLE
1.0
2.0
2.0
2.0
3.0
3.0
4.0
5.0
-
-

The following statement:

```
SELECT sample, COUNT(*) as number
   ...
GROUP BY sample;
```

Returns:

SAMPLE	NUMBER
1.0	1
2.0	3
3.0	2
4.0	1
5.0	1
-	2

Selecting Groups – the HAVING Clause

The `HAVING` clause restricts the selection of groups in the same way that a `WHERE` clause restricts the selection of rows. However, in contrast to the `WHERE` clause, a `HAVING` clause may use a set function on the left-hand side of a comparison.

The `HAVING` clause is most often used together with a `GROUP BY` clause, but may also be used to impose selection conditions on a column derived from a set function.

Find the currency codes that are used by more than one country:

```
SELECT currency_code, COUNT(currency_code)
FROM countries
GROUP BY currency_code
HAVING COUNT(currency_code) > 1;
```

Returns:

CURRENCY_CODE	
AUD	8
CHF	2
DKK	3
EUR	23
IDR	2
…	…

Ordering the Result Table

Strictly, the order of rows in a result table is undefined unless an `ORDER BY` clause is included in the `SELECT` statement.

Ascending or descending order may be specified; ascending order is the default.

Note: A `SELECT` statement without an `ORDER BY` clause may appear to give an ordered result in Mimer SQL, but you should include an `ORDER BY` clause if the ordering is important. Without the `ORDER BY` clause, a change in the database contents or updated Mimer SQL version can otherwise change the order.

Example

```
SELECT *
FROM formats
ORDER BY format DESC;
```

Returns:

FORMAT_ID	FORMAT	CATEGORY_ID	DISPLAY_ORDER
4	Vinyl	1	40
10	Video	3	20
8	Paperback	2	20
7	Hardcover	2	10
9	DVD Video	3	10
3	DVD Audio	1	10
2	Cassette	1	30
1	Audio CD	1	20
6	Audio CD	2	30
5	Audio Cassette	2	40

Ordering by More than One Column

More than one column may be specified in the `ORDER BY` clause.

Example

```
SELECT *
FROM formats
ORDER BY category_id, display_order;
```

Returns:

FORMAT_ID	FORMAT	CATEGORY_ID	DISPLAY_ORDER
3	DVD Audio	1	10
1	Audio CD	1	20
2	Cassette	1	30
4	Vinyl	1	40
7	Hardcover	2	10
8	Paperback	2	20
6	Audio CD	2	30
5	Audio Cassette	2	40
9	DVD Video	3	10
10	Video	3	20

Ordering by Set Function

To order a result table by a set function, the column in the result table is given a label and the label is used in the `ORDER BY` clause.

Example

```
SELECT category_id, MAX(display_order) AS maximum_value
FROM formats
GROUP BY category_id
ORDER BY maximum_value;
```

Returns:

CATEGORY_ID	MAXIMUM_VALUE
3	20
1	40
2	40

Ordering by a Computed Value

To order a result table by a computed value, place the computation in the `ORDER BY` clause.

Example

```
SELECT product
FROM products JOIN items ON products.product_id = items.product_id
WHERE format_id = 2
ORDER BY stock * price;
```

Returns:

PRODUCT
The Wild, the Innocent and the E Street Shuffle
Greatest Hits
On How Life Is
Snowed In
Christmas Portrait
Falling into You
LaTouché IV: Traditional Cajun Dancin' Music
Atlanta Homecoming
Born in the U.S.A.

Retrieving Data From More Than One Table

The examples presented up to now in this chapter have illustrated the essential features of simple `SELECT` statements with data retrieval from single tables. However, much of the power of SQL lies in the ability to perform joins through a single statement, i.e. to select data from two or more tables, using the search condition to link the tables in a meaningful way.

The Join Condition

In retrieving data from more than one table, the search condition or join condition specifies the way the tables are to be linked. For example:

List the product name in addition to the EAN and price:

```
SELECT product, ean_code, price
FROM items
    JOIN products ON items.product_id = products.product_id;
```

The join condition here is `ITEMS.PRODUCT_ID = PRODUCTS.PRODUCT_ID`, which relates the product identifier in table `ITEMS` (where codes are listed) to the product identifier in table `PRODUCTS` (where names are listed).

Returns:

PRODUCT	EAN_CODE	PRICE
100 Anos	77774238724	9.98
12 Golden Country Greats	75596190923	17.98
12 Super Exitos	724385487521	9.98
1492: Conquest of Paradise	75678243226	17.98
…	…	…

Conceptually, the join first establishes a table containing all combinations of the rows in PRODUCTS with the rows in ITEMS, then selects those rows in which the two PRODUCT_ID values are equal. See *Conceptual Description of the Selection Process* on page *70* for a fuller description of the conceptual SELECT process.

This does not necessarily represent the order in which the operations are actually performed; the order of evaluation of a complex SELECT statement is determined by the SQL optimizer, regardless of the order in which the component clauses are written.

Cross Products

Without the join condition, the result is a cross product of the columns in the tables in question, containing all possible combinations of the selected columns, for example:

```
SELECT product, ean_code, price
FROM items CROSS JOIN products;
```

Returns:

PRODUCT	EAN_CODE	PRICE
'Murder In The Cathedral'	77774238724	9.98
'Murder In The Cathedral'	75596190923	17.98
'Murder In The Cathedral'	724385487521	9.98
'Murder In The Cathedral'	75678243226	17.98
…	…	…

It is easy to see that a carelessly formulated join query can produce a very large result table. Two tables of 100 rows each, for instance, give a cross product with 10,000 rows; three tables of 100 rows each give a cross product with 1,000,000 rows!

The risk of generating large (erroneous) result tables is particularly high in interactive SQL (e.g. when using Mimer BSQL), where queries are so easily written and submitted.

Simple Joins

In simple joins, all tables used in the join are listed in the FROM clause of the SELECT statement. This is in distinction to nested joins, where the search condition for one SELECT is expressed in terms of another SELECT, see *Nested Selects* on page *56*.

Example

```
SELECT product, ean_code, price
FROM items
    JOIN products ON items.product_id = products.product_id;
```

SELECT *

The form `SELECT *` may be used in a join query, but since this selects all columns in the result set, at least one column is often duplicated (a join condition column).

Example

```
SELECT *
FROM items
    JOIN products ON items.product_id = products.product_id;
```

Returns:

Columns from `ITEMS`:

ITEM_ID	PRODUCT_ID	FORMAT_ID	RELEASE_DATE	STATUS	PRICE
STOCK	REORDER_LEVEL	EAN_CODE	PRODUCER_ID	IMAGE_ID	

Columns from `PRODUCTS`:

PRODUCT	PRODUCT_ID	PRODUCT_SEARCH

Columns in the join query that are uniquely identified by the column name may be specified by name alone. Columns that have the same name in the joined tables must be qualified by their respective table names.

The same query as above, but only three columns are returned:

```
SELECT product, ean_code, price
FROM items
    JOIN products ON items.product_id = products.product_id;
```

Nesting Join Clauses

It is possible to nest join-clauses, for example:

List the category in addition to the EAN and price for any items released in December 1996:

```
SELECT ean_code, price, category
FROM items
    JOIN formats ON items.format_id = formats.format_id
    JOIN categories ON categories.category_id = formats.category_id
WHERE release_date BETWEEN date'1996-12-01' AND date'1996-12-30';
```

Result:

EAN_CODE	PRICE	CATEGORY
9780006498957	7.99	Books
724385487521	9.98	Music
731453076723	29.98	Music
53308951925	11.98	Music

Complex Search Conditions and Joins

A join query can join any number of tables using complex search conditions to select the relevant information from each table.

List the product for any items with a release date in the future along with the item price in both Swedish and Danish crowns (SEK and DKK respectively):

```
SELECT product,
       CAST(price * exchange_rate AS DECIMAL(12,2)) AS cost,
       currency
FROM items
JOIN products ON products.product_id = items.product_id
CROSS JOIN currencies
WHERE release_date > CURRENT_DATE
  AND currencies.code IN ('SEK', 'DKK')
ORDER BY product, currency;
```

Result:

PRODUCT	COST	CURRENCY
Greatest Hits	99.42	Danish Kronor
Greatest Hits	125.61	Swedish Kronor
Pieces Of Fish	113.64	Danish Kronor
Pieces Of Fish	143.58	Swedish Kronor
The Future Foretold	49.71	Danish Kronor
The Future Foretold	62.80	Swedish Kronor
The Sql Quiz Book	99.49	Danish Kronor
The Sql Quiz Book	125.70	Swedish Kronor

In formulating a search condition for a join query, it can help to write out the columns that would appear in a complete cross-product of the tables. The search condition is then formulated as though the query was a simple `SELECT` from the cross-product table.

Outer Joins

The joins in the previous sections were all inner joins. In an inner join between two tables, only rows that fulfill the join condition are present in the result.

An outer join, on the contrary, contains non-matching rows as well. The outer join has two options, `LEFT` and `RIGHT`.

Left Outer Join

Example

```
SELECT ean_code, release_date, producer
FROM items
LEFT OUTER JOIN producers
    ON items.producer_id = producers.producer_id
WHERE ean_code >= 800000000000
ORDER BY ean_code;
```

Result:

EAN_CODE	RELEASE_DATE	PRODUCER
800488327626	1998-08-11	Giants Of Jazz (Ita)
801061007720	2000-10-31	Warp Records

EAN_CODE	RELEASE_DATE	PRODUCER
4988002364947	1999-09-28	-
4988011353147	1998-06-30	-
5013145800423	2000-03-14	Mint / Cherry Red
5013929112322	1999-10-12	Cherry Red
5014438710221	1994-12-27	Receiver Records
5019317001728	1994-12-15	Receiver Records
7157761806273	1996-01-18	Status Records
…	…	…

In the example above all rows from the table to the left in the join clause, i.e. `ITEMS`, are present in the result; non-matching rows from the `PRODUCERS` table are filled with null values in the result.

Observe the difference in result for the next statement and the previous one.

```
SELECT ean_code, release_date, producer
FROM items
LEFT OUTER JOIN producers
    ON  items.producer_id = producers.producer_id
    AND ean_code >= 800000000000
ORDER BY ean_code;
```

Result:

EAN_CODE	RELEASE_DATE	PRODUCER
8811038120	1991-08-27	-
8811042127	1991-10-22	-
8811061326	1992-05-19	-
8811067021	1992-12-22	-
…	…	…
800488327626	1998-08-11	Giants Of Jazz (Ita)
801061007720	2000-10-31	Warp Records
4988002364947	1999-09-28	-
4988011353147	1998-06-30	-
5013145800423	2000-03-14	Mint / Cherry Red
…	…	…

The reason is that conditions in the where clause are applied to the result of the join-clause and not to the joined tables as is the case with the conditions in the on-clause.

Right Outer Join

A right outer join will take all records from the table to the right in the join-clause.

Nesting Outer Joins

As with inner joins, it is possible to nest join-clauses. Nested joins can be of different types, i.e. both inner and outer joins.

The result of nested outer joins can be somewhat unexpected though, as it is the result of the first join-clause that is the left table in the next join, not the right table in the first join-clause.

Example

```
SELECT *
FROM tableA
    LEFT JOIN tableB ON tableA.id = tableB.id
    LEFT JOIN tableC ON tableA.id = tableC.id
```

This query does first perform `tableA LEFT JOIN tableB`. The result is then used as left table when performing `LEFT JOIN tableC`.

To make this query clearer, parentheses can be added as:

```
SELECT *
FROM (tableA LEFT JOIN tableB ON tableA.id = tableB.id)
  LEFT JOIN tableC ON tableA.id = tableC.id
```

Nested Selects

A form of `SELECT`, called a subquery, can be used in the search condition of a `SELECT` statement to form a nested query.

The main `SELECT` statement is then referred to as the outer select.

For example:

Select the products that have a release date in the future.

```
SELECT product
FROM products
WHERE product_id IN (SELECT product_id
                     FROM items
                     WHERE release_date > CURRENT_DATE);
```

Result:

PRODUCT
Greatest Hits
Pieces Of Fish
The Future Foretold
The Sql Quiz Book

To see how this works, evaluate the subquery first:

```
SELECT product_id
FROM items
WHERE release_date > CURRENT_DATE;
```

Result:

PRODUCT_ID
30206
30618

PRODUCT_ID
31082
31083

Then use the result of the subquery in the search condition of the outer select:

```
SELECT product
FROM products
WHERE product_id IN (30206, 30618, 31082, 31083);
```

Result:

PRODUCT
Greatest Hits
Pieces Of Fish
The Future Foretold
The Sql Quiz Book

Using Subqueries

A subquery can be used in a search condition wherever the result of the subquery can provide the correct form of the data for the search condition.

Thus a subquery used with '=' must give a single value as a result.

A subquery used with `IN`, `ALL` or `ANY` must give a set of single values, see *Retrieval with ALL, ANY, SOME* on page *62*.

A subquery used with `EXISTS` may give any result, see *Retrieving Data Using EXISTS and NOT EXISTS* on page *60*.

Examples:

```
WHERE column = (subquery)
WHERE column IN (subquery)
WHERE column = ALL (subquery)
WHERE column = ANY (subquery)
WHERE EXISTS (subquery)
```

The `UNION`, `EXCEPT` and `INTERSECT` operators can be used to combine two or more subqueries in more complex statements, see *Union, Except and Intersect Queries* on page *63*.

Nested Queries

Many nested queries can equally well be written as simple joins. For example:

Select the products that have a release date in the future.

```
SELECT product
FROM products
WHERE product_id IN (SELECT product_id
                     FROM items
                     WHERE release_date > CURRENT_DATE);
```

or alternatively

```
SELECT DISTINCT product
FROM products
JOIN items
    ON products.product_id = items.product_id
WHERE items.release_date > CURRENT_DATE;
```

Both these queries give exactly the same result. In most cases, the choice of which form to use is a matter of personal preference. Choose the form which you can understand most easily; the clearest formulation is least likely to cause problems.

Subqueries in Queries

Queries may contain any number of subqueries, for example:

List the producers (manufacturers) which have items that are more expensive than any of the items produced by Sony.

```
SELECT producer
FROM producers
WHERE producer_id IN
         (SELECT producer_id
          FROM items
          WHERE price >
                 (SELECT MAX(price)
                  FROM items
                  WHERE producer_id =
                          (SELECT producer_id
                           FROM producers
                           WHERE producer = 'SONY')));
```

Note the balanced parentheses for the nested levels.

It is particularly important at this level of complication to think carefully through the query to make sure that it is correctly formulated.

Often, writing some of the levels as simple joins can simplify the structure. The previous example may also be written:

```
SELECT DISTINCT producer
FROM producers
JOIN items
     ON producers.producer_id = items.producer_id
WHERE price > (SELECT MAX(price)
               FROM items
               JOIN producers
                   ON items.producer_id = producers.producer_id
               WHERE producer = 'SONY');
```

Correlation Names

A correlation name is a temporary name given to a table to represent a logical copy of the table within a query.

There are three uses for correlation names:

- simplifying complex queries
- joining a table to itself
- outer references in subqueries

Simplifying Complex Queries Using Correlation Names

Using short correlation names into complicated queries can make the query easier to write and understand, particularly when qualified table names are used:

```
SELECT mimer_store_music.artists.artist,
       mimer_store.product_details.*
FROM mimer_store.product_details
JOIN mimer_store_music.titles
    ON mimer_store.product_details.item_id =
       mimer_store_music.titles.item_id
JOIN mimer_store_music.artists
    ON mimer_store_music.artists.artist_id =
       mimer_store_music.titles.artist_id
ORDER BY mimer_store_music.artists.artist;
```

may be rewritten

```
SELECT art.artist, pdt.*
FROM mimer_store.product_details AS pdt
JOIN mimer_store_music.titles AS ttl
    ON pdt.item_id = ttl.item_id
JOIN mimer_store_music.artists AS art
    ON art.artist_id = ttl.artist_id
ORDER BY art.artist;
```

The keyword `AS` in the `FROM` clause may be omitted, but is recommended for clarity.

About Correlation Names

Correlation names are local to the query in which they are defined.

When a correlation name is introduced for a table name, all references to the table in the same query must use the correlation name.

The following expression is not accepted:

```
...
FROM mimer_store.product_details AS pdt,
     mimer_store_music.titles AS ttl,
...
WHERE ttl.item_id = mimer_store.product_details.item_id
```

Joining a Table with Itself Using a Correlation Name

Joining a table with itself allows you to compare information in a table with other information in the same table. This can be done with a correlation name.

Select all currencies with the same exchange rate:

```
SELECT c.currency, c.code, c.exchange_rate
FROM currencies AS c
JOIN currencies AS copy
    ON  c.exchange_rate = copy.exchange_rate
    AND c.currency <> copy.currency;
```

Result:

CURRENCY	CODE	EXCHANGE_RATE
Croatian Kuna	HRK	7.0820
Gourdes	HTQ	7.0820
Iraqi Dina	IQD	1551.0000
Uganda Shillings	UGX	1551.0000

Here, the table CURRENCIES is joined to a logical copy of itself called COPY.

The first search condition finds pairs of currencies with the same exchange rate, and the second eliminates 'pairs' with the same currency name. Without the second condition in the search criteria, all currencies would be selected!

Without correlation names, this kind of query cannot be formulated. The following query would select all the currencies from the table:

```
SELECT currency, code, exchange_rate
FROM currencies
WHERE currencies.exchange_rate = currencies.exchange_rate;
```

Outer References in Subqueries Using Correlation Names

In some constructions using subqueries, a subquery at a lower level may refer to a value in a table addressed at a higher level. This kind of reference is called an outer reference.

```
SELECT currency
FROM currencies
WHERE EXISTS (SELECT *
              FROM countries
              WHERE currency_code = currencies.code);
```

This kind of query processes the subquery for every row in the outer select, and the outer reference represents the value in the current outer select row. In descriptive terms, the query says 'For each row in CURRENCIES, select the CURRENCY column if there are rows in COUNTRIES containing the current CODE value'.

If the qualifying name in an outer reference is not unambiguous in the context of the subquery, a correlation name must be defined in the outer select.

A correlation name may always be used for clarity, as in the following example:

```
SELECT currency
FROM currencies AS c
WHERE EXISTS (SELECT *
              FROM countries
              WHERE currency_code = c.code);
```

Retrieving Data Using EXISTS and NOT EXISTS

EXISTS is used to check for the existence of some row or rows which satisfy a specified condition. EXISTS differs from the other operators in that it does not compare specific values; instead, it tests whether a set of values is empty or not. The set of values is specified as a subquery.

The subquery following the EXISTS clause most often uses of 'SELECT *' as opposed to 'SELECT column-list' since EXISTS only searches to see if the set of values addressed by the subquery is empty or not - a specified column is seldom relevant in the subquery.

EXISTS (subquery) is true if the result set of the subquery is not empty

NOT EXISTS (subquery) is true if the result set of the subquery is empty

SELECT statements with EXISTS almost always include an outer reference linking the subquery to the outer select.

Examples of EXISTS

Find all currencies that are used in the COUNTRIES table:

```
SELECT currency
FROM currencies AS c
WHERE EXISTS (SELECT *
              FROM countries
              WHERE currency_code = c.code);
```

Without the outer reference, the select becomes a conditional 'all-or-nothing' statement: perform the outer select if the subquery result is not empty, otherwise select nothing.

List all products where the producer (manufacturer) is not known:

```
SELECT product
FROM products AS p
WHERE EXISTS (SELECT *
              FROM items
              WHERE producer_id IS NULL
                AND product_id = p.product_id);
```

Examples of NOT EXISTS

The next example illustrates `NOT EXISTS`:

List all products where the producer (manufacturer) is not known:

```
SELECT product
FROM products
WHERE NOT EXISTS (SELECT *
                  FROM items
                  JOIN producers ON items.producer_id = producers.producer_id
                  WHERE product_id = products.product_id);
```

Result:

PRODUCT
Invictus
Middle Of Nowhere

Negated EXISTS

Negated `EXISTS` clauses must be handled with care. There are two semantic 'opposites' to `EXISTS`, with very different meanings:

```
WHERE EXISTS (SELECT *
              FROM artists
              WHERE artist = 'Enigma')
```

is true if at least one artist is called `Enigma`.

```
WHERE NOT EXISTS (SELECT *
                  FROM artists
                  WHERE artist = 'Enigma')
```

is true if no artist is called `Enigma`.

But

```
WHERE EXISTS (SELECT *
              FROM artists
              WHERE artist <> 'Enigma')
```

is true if at least one artist is not called `Enigma`.

```
WHERE NOT EXISTS (SELECT *
                  FROM artists
                  WHERE artist <> 'Enigma')
```

is true if no artist is not called `Enigma`, that is if every artist is called `Enigma`.

Retrieval with ALL, ANY, SOME

Subqueries that return a set of values may be used in the quantified predicates `ALL`, `ANY` or `SOME`. Thus

```
WHERE PRICE < ALL (subquery)
```

selects rows where the price is less than every value returned by the subquery

```
WHERE PRICE < ANY (subquery)
```

selects rows where the price is less than at least one of the values returned by the subquery

Select countries that have an exchange rate of less than one:

```
SELECT country
FROM countries
WHERE currency_code <> ALL (SELECT code
                            FROM currencies
                            WHERE exchange_rate >= 1.0);
```

If the result of the subquery is an empty set, `ALL` evaluates to true, while `ANY` or `SOME` evaluates to false.

An alternative to using `ALL`, `ANY` or `SOME` in a value comparison against a general subquery, is to use `EXISTS` or `NOT EXISTS` to see if values are returned by a subquery which only selects for specific values. For example:

Select countries where the associated currency code contains the letter 'E' as the middle character in the code:

```
SELECT country
FROM countries
WHERE currency_code = ANY (SELECT code
                           FROM currencies
                           WHERE code LIKE '_E_');
```

is equivalent to:

```
SELECT country
FROM countries AS c
WHERE EXISTS (SELECT *
              FROM currencies
              WHERE code LIKE '_E_'
                AND code = c.currency_code);
```

Union, Except and Intersect Queries

The UNION, EXCEPT and INTERSECT operators combine the results of two select clauses.

UNION first merges the result tables specified by the separate selects and then eliminates duplicate rows from the merged set. (UNION ALL does not eliminate duplicate rows.)

EXCEPT takes the distinct rows from the first select and returns the rows that do not appear in the second select. (EXCEPT ALL does not eliminate duplicate rows.)

INTERSECT takes the results of two selects and returns only rows that appear in both selects, after removing duplicate rows from the final result set. (INTERSECT ALL does not eliminate duplicate rows.)

Columns which are merged by UNION, EXCEPT and INTERSECT must have compatible data types (numerical with numerical, character with character, etc).

Subqueries addressing more than one result column are merged column by column in the order of selection. The number of columns addressed in each subquery must be the same.

The column names in the result of a UNION, EXCEPT or INTERSECT are taken from the names in the first subquery. Use labels in the first subquery to assign different column names to the result table.

In UNION, EXCEPT and INTERSECT queries, you may need to add an empty column so that columns not represented in both queries in the statement are retained in the result set. This is done by casting a null value to a matching data type.

Example

```
SELECT ean_code, release_date, producer
FROM items
INNER JOIN producers
    ON items.producer_id = producers.producer_id
UNION ALL
SELECT ean_code, release_date, CAST(NULL AS char)
FROM items
WHERE NOT EXISTS
    (SELECT * FROM producers
     WHERE items.producer_id = producers.producer_id)
```

UNION Examples

Select the different codes for currencies and countries that start with the letter 'D':

```
SELECT code
FROM currencies
WHERE code LIKE 'D%'
UNION
SELECT currency_code
FROM countries
WHERE country LIKE 'D%';
```

The result is obtained by merging the results of the two selects and eliminating duplicates:

```
SELECT code                    SELECT currency_code
FROM currencies                FROM currencies
WHERE code LIKE 'D%;           WHERE country LIKE 'D%';
```

CODE
DJF
DKK
DOP
DZD

CURRENCY_CODE
DJF
DKK
XCD
DOP

and the `UNION` gives the result table:

CODE
DJF
DKK
DOP
DZD
XCD

To retain duplicates in the result table, use UNION ALL in place of UNION, see the *Mimer SQL Reference Manual, Chapter 7, UNION or UNION ALL*, for details.

Merge the codes and names of currencies where the code begins with 'D' with the codes and names of the countries where the country begins with 'D':

```
SELECT code, currency AS currency_or_country
FROM currencies
WHERE code LIKE 'D%'
UNION
SELECT currency_code, country
FROM countries
WHERE country LIKE 'D%'
ORDER BY code;
```

Result:

CODE	CURRENCY_OR_COUNTRY
DJF	Djibouti
DJF	Djibouti Francs
DKK	Danish Kronor
DKK	Denmark
DOP	Dominican Pesos
DOP	Dominican Republic
DZD	Algerian Dinars
XCD	Dominica

Find the lowest and highest exchange_rates:

Unions can be used to combine information from the same table.

```
SELECT 'Highest', MAX(exchange_rate) AS rate
FROM currencies
UNION ALL
SELECT 'Lowest', MIN(exchange_rate)
FROM currencies
ORDER BY rate;
```

Result:

	RATE
Lowest	0.2644
Highest	1035000.0000

EXCEPT Example

Select the codes from currencies, except those that also are found in countries, starting with the letter 'D':

```
SELECT code
FROM currencies
WHERE code LIKE 'D%'
EXCEPT
SELECT currency_code
FROM countries
WHERE country LIKE 'D%';
```

The result is obtained by taking the first result and then remove the rows also found in the second select, and finally eliminating duplicates:

```
SELECT code
FROM currencies
WHERE code LIKE 'D%;
```

CODE
DJF
DKK
DOP
DZD

```
SELECT currency_code
FROM currencies
WHERE country LIKE 'D%';
```

CURRENCY_CODE
DJF
DKK
XCD
DOP

and the `EXCEPT` gives the result table:

CODE
DZD

To retain duplicates in the result table, use `EXCEPT ALL` in place of `EXCEPT`, see the *Mimer SQL Reference Manual, Chapter 7, EXCEPT or EXCEPT ALL*, for details.

INTERSECT Example

Select the codes from currencies and countries that exist in both tables, starting with the letter 'D':

```
SELECT code
FROM currencies
WHERE code LIKE 'D%'
INTERSECT
SELECT currency_code
FROM countries
WHERE country LIKE 'D%';
```

The result is obtained by taking the rows that are included in the first result and also in the second select, and finally eliminating duplicates:

```
SELECT code
FROM currencies
WHERE code LIKE 'D%;
```

CODE
DJF
DKK
DOP
DZD

```
SELECT currency_code
FROM currencies
WHERE country LIKE 'D%';
```

CURRENCY_CODE
DJF
DKK
XCD
DOP

and the `INTERSECT` gives the result table:

CODE
DFJ
DKK
DOP

To retain duplicates in the result table, use `INTERSECT ALL` in place of `INTERSECT`, see the *Mimer SQL Reference Manual, Chapter 7, INTERSECT or INTERSECT ALL*, for details.

Handling Null Values

Null values require special handling in SQL queries. Null represents an unknown value, and strictly speaking null is never equal to null. (Null values are however treated as equal for the purposes of `GROUP BY`, `DISTINCT` and `UNION`, `EXCEPT` and `INTERSECT`).

Searching for null

The condition for selecting null values is

```
WHERE column IS NULL
```

The negated form (`WHERE column IS NOT NULL`) selects values which are not null (i.e. values which are known).

List all currencies, and their codes, where the exchange rate is not known:

```
SELECT currency, code
FROM currencies
WHERE exchange_rate IS NULL;
```

Result:

CURRENCY	CODE
Armenian Drams	AMD
Kwanza	AOA
Brunei Dollars	BND
Francs Congolais	CDF
Saint Helena Pounds	SHP
Somali Shillings	SOS
Somoni	TJS

List all EAN codes where the producer is not known:

```
SELECT ean_code
FROM items
WHERE producer_id IS NULL;
```

Result:

EAN_CODE
4988002364947
4988011353147

List all EAN codes issued to Llewellyn Publications, where the release date is not known:

```
SELECT ean_code
FROM items
WHERE release_date IS NULL
AND PRODUCER_ID IN (SELECT producer_id
                    FROM producers
                    WHERE producer = 'Llewellyn Publications');
```

Result:

EAN_CODE
9780875428697
9780875428949
9780875428260
9780875428680
9780875427386

Null values in ALL, ANY, IN and EXISTS Queries

Null values should be treated cautiously, particularly in ALL, ANY, IN and EXISTS queries.

The result of a comparison involving null is unknown, which is generally treated as false. This can lead to unexpected results.

For example, neither of the following conditions are true:

```
<null>     IN (...,null,...)
<null> NOT IN (...,null,...)
```

The first result is almost intuitive: since null is not equal to null, null is not a member of a set containing null.

But if null is not a member of a set containing null, the second result is intuitively true.

In fact, neither result is true or false: both are unknown. If null values are involved on either side of the comparison, IN and NOT IN are not complementary. Similar arguments apply to queries containing ALL or ANY, for example:

```
SELECT currency, code
FROM currencies
WHERE exchange_rate > ALL (SELECT exchange_rate
                           FROM currencies
                           WHERE currency LIKE 'D%');
```

Result:

CURRENCY	CODE
Belarussian Rubles	BYR
Maticais	MZM
Lei	ROL
Turkish Liras	TRL

This query works as long as there are no null values returned by the subquery. But perform the subquery against a range of currencies that contain a null value in the exchange rate, and the query results in an empty set:

```
SELECT currency, code
FROM currencies
WHERE exchange_rate > ALL (SELECT exchange_rate
                           FROM currencies
                           WHERE currency LIKE 'A%');
```

Moreover, the reverse query, currencies that have a lower exchange rate, also results in an empty set. A justification for this is that as long as an exchange rate is unknown, it is impossible to say whether other currency rates are greater or less.

Using Exists

It is always possible to rephrase a query using ALL, ANY or IN in terms of one using EXISTS (with an outer reference between the selection and the EXISTS condition). This is to be recommended if the null indicator is to be permitted in the comparison sets, since null handling is then written out explicitly in the query.

Distinctions between queries involving null comparisons are subtle and are easily overlooked.

It is essential that the aim of a query is stringently defined before the query is formulated in SQL, and that the possible effects of null values in the search condition are considered.

There are many real-life examples where the presence of null has resulted in unforeseen and sometimes misleading data retrievals. It is advisable to define all columns in the database tables as `NOT NULL` except those where unknown values have a specific meaning. In this way the risks of confusion with null handling are minimized.

Conceptual Description of the Selection Process

This section presents a conceptual step-by-step analysis of the evaluation of a `SELECT` statement.

It is intended as an aid in formulating complex `SELECT` statements, and can also help you in understanding details of the statement syntax.

Note: The description here is purely conceptual. It does not represent the actual sequence of events performed by the database manager. In particular, the computer resource requirements implied by the intermediate result set defined in a `FROM` clause do not necessarily reflect actual requirements.

Query Used

The query used in the analysis is:

List those producers and the average price for the goods that they manufacture where they make more than 10 items. Sort the result by the average price, with the largest first:

```
SELECT producer, AVG(price) AS average
FROM producers AS p
JOIN items AS i
     ON p.producer_id = i.producer_id
GROUP BY p.producer
HAVING COUNT(*) > 10
ORDER BY average DESC, producer
```

Result:

PRODUCER	AVERAGE
BBC Audio (Spoken Word)	37.742727272727
MCA	27.798181818181
RCA	19.580000000000
Elektra/Asylum	18.265714285714
Warner Brothers	17.137894736842
Capitol	16.646666666666
Atlantic	14.798181818181
Sony	14.091111111111
…	…

Selection Process

Step 1 Subqueries at the lowest nesting level are evaluated first.

The first step in evaluating a select is to resolve subqueries from the lowest level up, and conceptually replace the subquery with the result set. The example here does not use a nested select.

When all subqueries are resolved, a, possibly complicated, single-level `SELECT` statement remains.

Step 2 The FROM clause defines an intermediate result set.

Tables addressed in the `FROM` clause are combined to form an intermediate result set which is the full cross product of the tables.

The cross product is a table with one column for each column in each of the table, and one row for every combination of rows from the different tables.

The columns in the result set are identified by the qualified column names from the table from which they are derived.

```
FROM producers AS p JOIN items AS i
```

The `FROM` clause in the example produces an intermediate result set which is the full cross product of the `PRODUCERS` table and the `ITEMS` table.

Step 3 The ON clause selects rows from the intermediate set.

The `ON` clause selects rows from the full cross product result set that meet the `JOIN` criteria specified.

```
ON p.producer_id = i.producer_id
```

In this example the `ON` clause selects only those result set rows where the value in the `PRODUCER_ID` column from the `PRODUCERS` table is equal to that in the `PRODUCER_ID` column from the `ITEMS` table.

The GROUP BY clause groups the remaining result set:

```
GROUP BY p.producer
```

PRODUCER	PRICE
404 Music Group	16.98
4AD Records	11.98
7-N Music	16.98
A&M Records	11.98
A&M Records	22.98
A&M Records	10.98
A&M Records	18.98
A&M Records	18.98
…	…

Step 4 The HAVING clause selects groups:

```
HAVING COUNT(*) > 10
```

PRODUCER	PRICE
Atlantic	17.98
Atlantic	11.98
Atlantic	11.98
Atlantic	9.98
Atlantic	11.98
Atlantic	17.98
Atlantic	11.98
Atlantic	11.98
…	…

Step 5 The SELECT list selects columns, evaluates any expressions in the SELECT list, and reduces groups to single rows if set functions are used:

```
SELECT producer, AVG(price) AS average
```

PRODUCER	AVERAGE
Atlantic	14.798181818181
BBC Audio (Spoken Word)	37.742727272727
Capitol	16.646666666666
Collins	7.529814814814
Elektra/Asylum	18.265714285714
Geffen Records	12.480000000000
HarperCollins	6.722187500000
Marshall Editions	9.842222222222
…	…

Step 6 The results of subqueries joined by UNION, EXCEPT and INTERSECT are merged.

This example does not include a UNION, EXCEPT or INTERSECT.

Step 7 The final result is sorted according to the ORDER BY clause:

```
ORDER BY average DESC, producer;
```

PRODUCER	AVERAGE
BBC Audio (Spoken Word)	37.742727272727
MCA	27.798181818181
RCA	19.580000000000
Elektra/Asylum	18.265714285714
Warner Brothers	17.137894736842

PRODUCER	AVERAGE
Capitol	16.646666666666
Atlantic	14.798181818181
Sony	14.091111111111
…	…

Chapter 4

Collations

Sorting and searching non-English text can cause a number of problems, a frequent one being how to handle accented letters, for example á, à and â.

The rules for sorting vary because the various natural languages sort words differently. There are occasions where the accented form of a letter is treated as a distinct letter for the purpose of comparison. For example, in Swedish, Å is a separate letter that is sorted after Z. In some languages, it is common to sort uppercase before lowercase, in other languages this is reversed; sometimes it is just a matter of personal preference.

A collation, also known as a collating sequence, is a database object containing a set of rules that determines how character strings are compared, searched and alphabetically sorted. The rules in the collation determine whether one character string is less than, equal to or greater than another. A collation also determines how case-sensitivity and accents are handled.

In Mimer SQL, a collation belongs to a schema. In this release, the pre-defined collations included belong to `INFORMATION_SCHEMA`.

When a collation is used, Mimer SQL first checks to see if it belongs to the ident's schema. If Mimer SQL does not find it there, it checks for it in `INFORMATION_SCHEMA`.

Character Sets and Collations

For character data, Mimer SQL uses the character set ISO 8859-1, also known as the Latin1 character set. By default, character data is sorted in the numerical order of their codes according to the `ISO8BIT` collation.

For national character data, Mimer SQL uses the Unicode character set. By default national character data is sorted according to the `UCS_BASIC` collation.

It is not possible to add, drop or modify a character set.

Every character set has one default collation.

Character Set	Default Collation	Data Types
ISO 8859-1	ISO8BIT	CHARACTER CHARACTER VARYING CLOB
UNICODE	UCS_BASIC	NCHAR NCHAR VARYING NCLOB

If you want to sort characters in a different way than the default, you can specify a collation at the column level when creating or altering a table or creating a domain. You can also override a collation by using a `COLLATE` clause in an SQL statement.

Using Collations

You can specify a collation for ordering characters when you create or alter a table or create a domain.

If you have specified a collation for a column, the collation is used implicitly in SQL statements.

You only need to explicitly use a collate clause in SQL statements if you want to override the default collation or the collation you specified when creating or altering the table or creating the domain.

Character Strings

SQL only permits compatible character strings to be compared. That is, you can compare character strings only if the source and target strings belong to the same collation or can be coerced into having the same collation.

A character string that is defined with a named collation can only be compared or assigned to a character string that is either defined with the same named collation or is defined without a collation.

In the case where one of the strings is not associated with a named collation then it will be implicitly coerced to the same collation as the other string.

String Comparison Examples

The following three comparisons are all legal (and equivalent):

```
job_title = 'developer' COLLATE english_1

job_title COLLATE english_1 = 'developer'

job_title COLLATE english_1 = 'developer' COLLATE english_1
```

But –

```
job_title COLLATE english_1 = 'developer' COLLATE swedish_1
```

is illegal because different collations are specified.

CREATE/ALTER TABLE

When creating or altering a table, you can specify a collation in the column-definition, for example:

```
CREATE TABLE employees (surname CHAR(20) COLLATE swedish_1
            ...
```

CREATE DOMAIN and CREATE TYPE

When creating a domain you can specify a collation for the character and national character string data types, for example:

```
CREATE DOMAIN name_type AS VARCHAR(48) COLLATE english_1;
CREATE TYPE car_models AS VARCHAR(48) COLLATE english_1;
```

All properties of a domain apply to the column when the domain/type is used in a `CREATE TABLE` or `ALTER TABLE` statement.

CREATE INDEX

To improve performance when retrieving data, you can create more than one index for a column using different collations, for example:

```
CREATE INDEX cnt_eng_ind ON countries (country COLLATE english_3);
CREATE INDEX cnt_swe_ind ON countries (country COLLATE swedish_3);
```

Which index that will be used depends on the situation. For example:

```
SELECT * FROM countries ORDER BY country COLLATE english_3;
```

will use the `cnt_eng_ind` index.

And

```
SELECT * FROM countries ORDER BY country COLLATE swedish_3;
```

will use the `cnt_swe_ind` index.

Collation Precedence

A collation specified in the column-definition will take precedence over a domain collation.

Continuing with the example above, the domain collation was set to `english_1`, but in the following example the column `country` is set to `swedish_1`, which takes precedence over the domain setting:

```
CREATE TABLE countries (
code CHARACTER(2),
country name_type COLLATE swedish_1,
...
```

Altering Collations on Columns

You can change the collation specified for a column by using the `ALTER TABLE` statement, for example:

```
ALTER TABLE countries ALTER COLUMN country CHAR(20) COLLATE english_1;
```

To return to the default (`ISO8BIT`) sorting order, you would enter:

```
ALTER TABLE countries ALTER COLUMN country CHAR(20) COLLATE ISO8BIT;
```

By altering a collation, for example to the default `ISO8BIT` collation, you can remove any dependencies associated with the collation. This makes it possible to drop the collation – see the next section.

Dropping a Collation

You can drop a collation only if there are no dependencies, for example:

```
DROP COLLATION collation_name RESTRICT;
```

Finding Out the Default Collation For a Column

You can find out which collation a column uses by reviewing the `INFORMATION_SCHEMA.COLUMNS` view, for example:

```
SELECT *
FROM INFORMATION_SCHEMA.COLUMNS
WHERE table_name = 'table1'
  AND column_name = 'col1';
```

For more information, see the *Mimer SQL Reference Manual, Chapter 13, INFORMATION_SCHEMA dictionary views*.

Using Collations – Examples

The following sections contain examples of how to use collations and what effects collations can have on a result set.

The examples are based on the following (rather simple) table, `table1`:

colswe	coleng	col1
A	A	A
a	a	a
W	W	W
å	å	å
v	v	v

`colswe` uses the `swedish_1` collation, `coleng` uses the `english_1` collation and `col1` uses the Mimer SQL default `ISO8BIT` collation.

Comparison Operators

You can qualify the comparison operators (=, <>, <, <=, …) with a `COLLATE` clause. For example:

```
SELECT col1
FROM table1
WHERE col1 > 'm';
```

would give the following result:

col1
å
v

However, explicitly using the COLLATE clause and the english_1 collation:

```
SELECT col1
FROM table1
WHERE col1 > 'm' COLLATE english_1;
```

would give the following result:

col1
W
v

Similarly, explicitly using the COLLATE clause and the swedish_1 collation:

```
SELECT col1
FROM table1
WHERE col1 > 'm' COLLATE swedish_1;
```

would give the following result:

col1
W
å
v

ORDER BY

You can use a COLLATE clause together with an ORDER BY clause to sort result sets. In most cases a level 3 collation is suitable for order by purposes. For example:

```
SELECT *
FROM table1
ORDER BY col1 COLLATE swedish_3;
```

retrieves the data and sorts it on col1 according to the swedish_3 collation:

colswe	coleng	col1
a	a	a
A	A	A
v	v	v
W	W	W
å	å	å

Similarly, the following statement:

```
SELECT *
FROM table1
ORDER BY col1 COLLATE english_3;
```

retrieves the data and sorts it according to the english_3 collation:

colswe	coleng	col1
a	a	a
A	A	A

colswe	coleng	col1
å	å	å
v	v	v
W	W	W

Note: Where a collation defines a number of characters with the same sort-order value, the retrieval order within the sort-order value is not defined.

GROUP BY

Depending on the collation associated with a column, you might get differing results when using `GROUP BY`.

For example, the statement:

```
SELECT col1, COUNT(*)
FROM table1
GROUP BY col1 COLLATE swedish_1;
```

gives the following result:

col1	
A	2
W	2
å	1

According to the `swedish_1` collation, two instances of the character ‘a’ were found and one instance of ‘å’ which is considered a separate character in the Swedish language.

Similarly, using the `english_1` collation in the statement:

```
SELECT col1, COUNT(*)
FROM table1
GROUP BY col1 COLLATE english_1;
```

gives the following result:

col1	
A	3
v	1
W	1

According to the `english_1` collation, three instances of the character ‘a’ were found, as the character ‘å’ has the same sort-order value as ‘A’ and ‘a’.

Scalar String Functions

You can use the `COLLATE` clause with the scalar string functions `SUBSTRING` and `TRIM`.

Character strings that are derived from a single string, for example, those returned from the `TRIM` and `SUBSTRING` functions, inherit the collation from the source string.

TRIM and COLLATE

You should be aware of the consequences when you use a `TRIM` function on a column that has a collation.

For example, referring to `table1`, see *Using Collations – Examples* on page *78*, the following statement:

```
SELECT TRIM('v' FROM colswe)
FROM table1;
```

would trim both 'W' and 'v' from the result set as the characters 'W' and 'v' have the same sort-order value in a Swedish case-insensitive collation.

Similarly, the following statement:

```
SELECT TRIM('a' FROM col1)
FROM table1;
```

would trim 'A', 'a' and 'å' from the result set as the characters 'A', 'a' and 'å' have the same sort-order value in an English case-insensitive collation.

Concatenation Operator

Suppose you want to concatenate columns, `colswe` and `col1`, for example:

```
SELECT colswe || coleng
FROM table1;
```

Because the columns use different collations the result set will have the default collation `ISO8BIT`.

However, if you want apply a collation to the result set, you can add a `COLLATE` clause. for example:

```
SELECT (colswe || coleng) COLLATE swedish_1
FROM table1;
```

IN and BETWEEN

A collation will affect the results of a query that uses `IN` or `BETWEEN`.

For example, the following statement:

```
SELECT *
FROM table1
WHERE coleng BETWEEN 'a' and 'B';
```

returns:

colswe	coleng	col1
A	A	A
a	a	a
å	å	å

But, the statement:

```
SELECT *
FROM table1
WHERE colswe BETWEEN 'a' and 'B';
```

returns

colswe	coleng	col1
A	A	A
a	a	a

UNION, EXCEPT and INTERSECT

When performing a `UNION` (or `EXCEPT` or `INTERSECT`), you must know what collations are involved in order to ensure that you get the result you want.

For example, the following statement:

```
SELECT colswe
FROM table1

UNION

SELECT coleng
FROM table1;
```

raises an error because the `UNION` operator can't understand which duplicate rows to remove or not.

To perform the `UNION` according to the `swedish_1` collation, you would explicitly use a `COLLATE` clause, for example:

```
SELECT colswe COLLATE swedish_1
FROM table1

UNION

SELECT coleng
FROM table1;
```

which would return:

colswe
A
W
å

Similarly, for a UNION result according to the english_1 collation, you would enter:

```
SELECT colswe COLLATE english_1
FROM table1

UNION

SELECT coleng
FROM table1;
```

which would return:

colswe
A
v
W

DISTINCT

When you use DISTINCT, you must consider the consequences of which collation is associated with a column.

In the following example:

```
SELECT DISTINCT col1
FROM table1;
```

All entries in col1 are considered DISTINCT as it uses the Mimer SQL default collation ISO8BIT:

col1
A
W
a
v
å

However, in this next statement:

```
SELECT DISTINCT colswe
FROM table1;
```

colswe uses the swedish_1 collation. 'å' and 'A' are considered to be distinct, but 'v' and 'W' are not:

colswe
A
W
å

Similarly, in this example:

```
SELECT DISTINCT coleng
FROM table1;
```

`coleng` uses the `english_1` collation, ‘v’ and ‘W’ are considered to be distinct, but ‘å’ and ‘A’ are not:

coleng
A
v
W

Chapter 5

Working With Data

This chapter deals with manipulating the data in tables with the statements:

- `INSERT` for inserting new rows into tables
- `UPDATE` for updating rows
- `DELETE` for deleting rows from tables
- `CALL` for manipulating data by executing procedures.

Access Privileges

You must have the appropriate access privileges on the relevant table(s) in order to use `INSERT`, `UPDATE` or `DELETE`.

In addition, the table itself must be updatable. All base tables are updatable, but some views are not, see *Updatable Views* on page *90*.

In order to make a `CALL` you must have `EXECUTE` privilege on the procedure.

Inserting Data

The `INSERT` statement is used to insert new rows into existing tables.

Values to be inserted may be specified explicitly, as constants or expressions, or in the form of a subquery, see below.

The data to be inserted must be of a type compatible with the corresponding column definition.

If the length of the inserted data differs from that of the column definition, the data is handled as follows:

Data	Explanation
Character strings	If the inserted data is longer than the column definition, an error is reported and the `INSERT` operation fails (trailing spaces are truncated without error). If the inserted data is shorter than the column definition, it is padded to the right with spaces to the required length when inserted into a fixed-length character column. The inserted data is not padded when inserted into a `VARCHAR` or `NCHAR VARYING` column.

Data	Explanation
Decimal values	Decimal values which are longer than the column definition are truncated (not rounded) from the right to meet the column definition. Thus `12.3456` is inserted into `DECIMAL(6,3)` as `12.345`. Decimal values which are shorter than the column definition are padded to the right of the decimal point with zeros. Thus `12.3` is inserted into `DECIMAL(6,3)` as `12.300`.
Integer values	If the inserted data has more digits than the column definition or is outside the range of the definition, an error is reported and the `INSERT` operation fails.
Floating point values	Floating point values are converted to decimal by truncating the fractional part of the value as required by the scale of the decimal target. An error occurs if the scale of the target cannot accommodate the integral part of the value.
Datetime values	Date values are converted to timestamp by setting the hour, minute and second fields to zero. Time values are converted to timestamp by taking values for the year, month and day fields from `CURRENT_DATE`. Timestamp values are converted to date or time by discarding the field values that do not appear in the target.
Interval values	Single field interval values are converted to exact numeric by truncating decimal digits or by padding decimal digits with zeros. If any loss of leading precision occurs, or if overflow occurs, an error is raised.
Binary values	If the inserted data is longer than the column definition, an error is reported and the `INSERT` operation fails. If the inserted data is shorter than the column definition, and the column is fixed-length binary, an error is reported and the `INSERT` operation fails.

Inserting Explicit Values

The explicit `INSERT` statement has the general form:

```
INSERT INTO table [(column-list)]
   VALUES (value-list);
```

Values in the value-list are inserted into columns in the column-list in the order specified.

The order of columns in the column-list need not be the same as the order in the table definition. Any columns in the table definition which are not included in the column-list are assigned null values, or the column default value if one is defined.

An explicit `INSERT` statement can only insert a single row.

For example:

Insert the values 'GW', 'Guinea-Bissau' and 'XOF' into the CODE, COUNTRY and CURRENCY_CODE columns respectively into the COUNTRIES table:

```
INSERT INTO countries(code, country, currency_code)
   VALUES ('GW', 'Guinea-Bissau', 'XOF');
```

inserts the row:

CODE	COUNTRY	CURRENCY_CODE
GW	Guinea-Bissau	XOF

If you insert explicit values into all of the columns in a table, the column list can be omitted from the `INSERT` statement. The values specified are then inserted into the table in the order that the columns are defined in the table.

Thus the example above could also be written:

```
INSERT INTO countries
   VALUES ('GW', 'Guinea-Bissau', 'XOF');
```

Inserting Results of Expressions

You can also insert the result of an expression into a table:

```
INSERT INTO mimer_store.customers(customer_id,
                                  title, surname, forename,
                                  date_of_birth,
                                  address_1, address_2, town,
                                  postcode, country_code,
                                  email, password,
                                  registered)
   VALUES (DEFAULT,
           'Mr', 'Eriksson', 'Sven',
            mimer_store.cast_to_date('25/10/1953'),
           'Kungsgaten 64', 'Box 1713', 'Uppsala',
           '751 47', 'SE',
           'training@mimer.com', 'secret',
           CURRENT_DATE);
```

Inserting with a Subquery

Values to be inserted can also be specified in the form of a subquery, i.e. fetched from a table in the database.

```
INSERT INTO formats
  SELECT 11, 'Book & Cassette', MAX(formats.category_id),
         MAX(display_order) + 10
  FROM formats JOIN categories
      ON formats.category_id = categories.category_id
  WHERE category = 'Books';
```

Inserting the result of a subquery can insert a number of rows into a table. If any of the rows are rejected (e.g. because of a duplicate primary or unique key), the whole `INSERT` statement fails and no rows are inserted.

Inserting Sequence Values

The value to be inserted can be the value of a sequence. The constructs that return the current value or next value of a sequence can be used as column values in the INSERT statement:

```
INSERT INTO products(product, product_id)
   VALUES ('SQL Reference', NEXT VALUE FOR product_id_seq);

INSERT INTO mimer_store_music.titles(item_id, artist_id)
   VALUES (CURRENT VALUE FOR mimer_store.item_id_seq, 500999);
```

Inserting Null Values

The keyword NULL may be used to insert the null value into a column, provided that the column is not defined as NOT NULL:

```
INSERT INTO tracks(item_id, track_no, track, length)
   VALUES (60099, 14, 'Bayamesa', NULL);
```

The null indicator is implicitly inserted into columns when no value is given for that column and the column definition does not include a default value.

Thus, the following INSERT statement will give the same results as the example above:

```
INSERT INTO tracks(item_id, track_no, track)
   VALUES (60099, 14, 'Bayamesa');
```

Updating Tables

Data in existing table rows can be changed with the UPDATE statement. This statement has the general form:

```
UPDATE table
   SET column = value
   WHERE search-condition;
```

The search condition specifies which rows in the table are to be updated. If no search condition is specified, all rows will be updated.

Update the exchange rate for US dollars to 0.8886:

```
UPDATE CURRENCIES
   SET EXCHANGE_RATE = 0.8886
   WHERE CODE = 'USD';
```

Discount all Sony prices by 10 percent:

```
UPDATE items
   SET price = price * 0.90
   WHERE producer_id IN (SELECT producer_id
                         FROM producers
                         WHERE producer = 'Sony');
```

Primary key columns can be updated provided the table is stored in a databank with TRANSACTION or LOG option.

Deleting Rows from Tables

The DELETE statement removes rows from a table, and has the general form:

```
DELETE FROM table
WHERE search-condition;
```

The search condition specifies which rows in the table are to be deleted. If no search condition is specified, all rows will be deleted (the table is emptied but not dropped).

Delete all countries that begin with the letter 'D' from the COUNTRIES table:

```
DELETE FROM countries
WHERE country LIKE 'D%';
```

Delete all rows from the CUSTOMERS table:

```
DELETE FROM customers;
```

Delete all Sony items:

```
DELETE FROM mimer_store.items
WHERE producer_id IN (SELECT producer_id
                      FROM mimer_store.producers
                      WHERE producer = 'Sony');
```

Calling Procedures

In addition to using data manipulation statements directly, as just described, it is also possible to manipulate table data by calling a procedure. Procedures perform the specific data manipulations laid out in the procedure definition.

Any SQL statement in the grouping procedural-sql-statement, see the *Mimer SQL Reference Manual, Chapter 12, Procedural SQL Statements*, can be used in a procedure, and this includes all the data manipulation statements.

The use of procedures allows data manipulation within the database to be controlled both in terms of strictly defining which data manipulation operations are performed and also in terms of regulating which database objects can be affected.

In the CALL statement, the value-expressions or assignment targets specified for each of the procedure parameters must be of a data type that is assignment-compatible, see the *Mimer SQL Reference Manual, Chapter 7, Assignments*, with the parameter data type.

See the *Mimer SQL Reference Manual, Chapter 12, CALL*, for full details of the CALL statement and the *Mimer SQL Programmer's Manual, Chapter 11, Mimer SQL Stored Procedures*, for a general discussion of the stored procedure functionality supported in Mimer SQL.

Examples of Calling Procedures

Invoke the procedure called SEARCH in the MIMER_STORE_MUSIC schema:

```
CALL mimer_store_music.search(:title, :artist, CAST(NULL as integer));
```

Updatable Views

INSERT, UPDATE and DELETE statements may be used on views.

The operation is then performed on the base table upon which the view is defined. However, certain views may not be updated (for example a view containing DISTINCT values, where a single row in the view may represent several rows in the base table).

A view is not updatable if any of the following conditions are true:

- the keyword DISTINCT is used in the view definition
- the select list contains components other than column specifications, or contains more than one specification of the same column
- the FROM clause specifies more than one table reference or refers to a non-updatable view
- the GROUP BY clause is used in the view definition
- the HAVING clause is used in the view definition

Note: By defining an INSTEAD OF trigger any view can be made updatable. If all the INSTEAD OF triggers on the view are dropped, the view will revert to not updatable if one or more of the above conditions are true.

Chapter 6

Managing Transactions

This chapter discusses transaction principles, logging and handling transactions.

Transaction Principles

A transaction is an environment where it is possible to `COMMIT` all of the operations performed within it, or to ensure that all of them fail.

Transaction Phases

In general, three transaction phases exist:

- build-up, during which the database operations are requested
- prepare, during which the transaction is validated
- commitment, during which the operations performed in the transaction are written to disk.

Transaction build-up, which may be started explicitly or implicitly; prepare and commitment are both initiated explicitly through a request to commit the transaction (using `COMMIT`).

In interactive application programs, build-up takes place typically over a time period determined by the user, while prepare and commitment are part of the internal process of committing a transaction, which occurs on a time-scale determined by machine operations.

The transaction begins by taking a snapshot of the database in a consistent state.

During build-up, changes requested to the contents of the database are kept in a write-set and are not visible to other users of the system. This allows the database to remain fully accessible to all users. The application program in which build-up occurs will see the database as though the changes had already been applied. All changes requested during transaction build-up become visible to other users when the transaction is successfully committed.

A major function of the transaction handling in Mimer SQL multi-user systems is concurrency control. This means protecting the database from inconsistency which might arise when two users attempt to change the same information at the same time.

Mimer SQL supports distributed transactions based on the XA interface as defined by the Open Group and Microsoft's Distributed Transaction Coordinator (DTC) protocol. This means that Mimer SQL can be used in application environments that support distributed transactions.

See the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security*, for a more detailed discussion of transaction handling and database security.

Logging Transactions

Transaction control also provides the basis for protection of the database against hardware failure.

Changes made to a database may be logged, to provide back-up protection in the event of hardware failure, provided that the changes occur within a transaction and that the databanks involved have the `LOG` option. Transaction handling is, therefore, important even in standalone environments where concurrency control issues do not arise.

The system logging databank, `LOGDB` is where transaction changes are recorded. It contains a record of all transactions executed since the latest back-up copy of a databank was taken and the log cleared. The latest back-up copy of the databank, together with the contents of `LOGDB`, may be used to restore the databank in the event of a databank crash.

Logging Options

Transaction control and logging is determined at the databank level by options set when the databank is defined.

The options are:

Option	Description
`LOG`	All operations on the databank are performed under transaction control. All transactions are logged.
`TRANSACTION`	All operations on the databank are performed under transaction control. No transactions are logged.
`WORK`	All operations on the databank are performed without transaction control (even if they are requested within a transaction), and are not logged. Sets of operations (`DELETE`, `UPDATE` or `INSERT` on several rows) which are interrupted will not be rolled back.
`READ ONLY`	Only read only operations are allowed, i.e. `DELETE`, `UPDATE` or `INSERT` can not be performed on tables in a databank with this option.

All important databanks should be defined with `LOG` option, so that valuable data is not lost by any system failure.

Handling Transactions

Transaction control statements in Mimer SQL are:

- COMMIT
- COMMIT BACKUP
- ROLLBACK
- SET TRANSACTION READ ONLY
- SET TRANSACTION READ WRITE
- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
- SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
- SET TRANSACTION ISOLATION LEVEL READ COMMITTED
- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
- SET TRANSACTION START EXPLICIT
- SET TRANSACTION START IMPLICIT
- SET TRANSACTION DIAGNOSTICS SIZE
- SET SESSION CHARACTERISTICS AS TRANSACTION READ ONLY
- SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
- SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE
- SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL REPEATABLE READ
- SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED
- SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
- SET SESSION CHARACTERISTICS AS TRANSACTION DIAGNOSTICS SIZE
- START BACKUP
- START TRANSACTION

SQL Statement Restrictions in Transactions

The following SQL statements may not be used inside a transaction:

ENTER	SET SHADOW
LEAVE	SET TRANSACTION
SET DATABANK	START BACKUP
SET DATABASE	START TRANSACTION
SET SESSION	

Data definition statements (e.g. ALTER, DROP, UPDATE STATISTICS) may be used inside a transaction provided they are the only statement executed in that transaction.

Optimizing Transactions

It is strongly recommended that the SET TRANSACTION READ ONLY option be used for each transaction that does not perform updates to the database and that the SET TRANSACTION READ WRITE option be used only when a transaction performs updates.

Taking a little extra care to set these options appropriately will ensure the transaction performance remains optimal at all times.

The default transaction read option can be defined by using SET SESSION, see *Default Transaction Options* on page *94*. If this has not been used to set the default transaction read option, the default is READ WRITE.

Consistency Within a Transaction

The SET TRANSACTION ISOLATION LEVEL options are provided to control the degree to which the updates performed by one transaction are affected by the updates performed by other transactions which are executing concurrently.

The default isolation level can be defined by using SET SESSION, see *Default Transaction Options* on page *94*. If this has not been used to set a default isolation level, the default is REPEATABLE READ. This isolation level guarantees that the end result of the operations performed by two or more concurrent transactions is the same as if the transactions had been executed in a serial fashion, except that an effect known as 'Phantoms' may occur.

This is where one transaction reads a set of rows that satisfy some search condition. Another transaction then performs an update which generates one or more new rows that satisfy that search condition. If the original query is repeated (using exactly the same search condition), extra rows appear in the result set that were previously not found.

The other isolation levels are: READ UNCOMMITTED, READ COMMITTED and SERIALIZABLE.

All four isolation levels guarantee that each transaction will be executed completely or not at all and that no updates will be lost.

Refer to *Mimer SQL Reference Manual, Chapter 12, SET TRANSACTION*, for a full description of the effects that are possible, or guaranteed never to occur, at each of the four isolation levels.

Default Transaction Options

- **SET SESSION**

  The SET SESSION statement is provided so that default values for certain transaction control settings can be defined.

  SET SESSION allows the default settings for SET TRANSACTION READ and SET TRANSACTION ISOLATION LEVEL to be defined.

- **SET TRANSACTION READ and SET TRANSACTION ISOLATION LEVEL**

  The transaction control settings defined by SET TRANSACTION READ, see *Optimizing Transactions* on page *94*, and SET TRANSACTION ISOLATION LEVEL, see *Consistency Within a Transaction* on page *94*, apply to the single next transaction to be started. If these statements are not used explicitly before each transaction, the default settings apply.

Chapter 7

Creating a Database

This chapter describes the SQL statements for creating and managing the database structure. Examples are based on the database listed in *Appendix B The Example Environment*.

In addition, Mimer BSQL provides specific commands for listing and describing database objects, see *Chapter 9, Mimer BSQL*.

SQL includes statements for creating and modifying the database structure:

- create idents, schemas, databanks, shadows, domains, sequences, tables, triggers, functions, procedures, modules, views, indexes and synonyms
- saving documentary comments on objects
- altering the definition of idents, databanks, shadows and tables
- dropping objects from the database.

All information describing the database structure is stored in the data dictionary.

Database Modelling

Before the database is defined, it is extremely important to design the database model. Well-functioning and efficient databases cannot be created without a model as the foundation.

Without careful design, much of the flexibility and efficiency inherent in a relational database structure may be lost.

Creating Idents and Schemas

Idents are authorized users of the system or groups of users defined for easier ident management, see *Idents* on page *16*.

A schema defines a local environment within which private database objects can be created. The ident creating the schema has the right to create objects in it and to drop objects from it.

The statement for creating idents has the general form:

```
CREATE IDENT username
AS ident-type
[USING 'password']
[WITH | WITHOUT   SCHEMA];
```

Ident Names

The case of letters is insignificant for an ident name and it must be composed of a unique sequence of case-less characters (e.g. the idents `ABC` and `aBc` cannot both exist in the database because they are identical when case is ignored).

Passwords

Passwords are composed of case-significant characters and must be entered exactly as they are defined.

Passwords are optional for `USER` idents. A `USER` ident with an `OS_USER` login may connect to Mimer SQL without providing a password. Passwords are required for `PROGRAM` idents. Passwords are not used for `GROUP` idents.

Schemas

When a `USER` or `PROGRAM` ident is created, a schema with the same name can also be created automatically and the created ident becomes the creator of the schema. This happens by default unless `WITHOUT SCHEMA` is specified in the `CREATE IDENT` statement. For idents who are not supposed to create database objects, it's good practice to specify `WITHOUT SCHEMA`.

All private database objects created by an ident must belong to a schema which, by default, is the schema with the same name as the ident. When any private database object is created, its name can be specified in the fully qualified form that explicitly identifies which schema the object is to belong to. An ident may create objects in schemas 'owned' by it (i.e. the schema created automatically when the ident was created and any schemas explicitly created by the ident).

An ident with `IDENT` or `SCHEMA` privilege can create additional schemas by using the `CREATE SCHEMA` statement. The objects belonging to the schema can be defined in the `CREATE SCHEMA` statement and created at the same time as the schema, refer to the *Mimer SQL Reference Manual, Chapter 12, CREATE SCHEMA* for details.

Creating Idents and Schemas, Examples

Create a user ident MIMER_ADM with the password 'adm':

Note: Schema `MIMER_ADM` will also be automatically created.

```
CREATE IDENT mimer_adm AS USER USING 'adm';
```

Create a program ident AUDIT with the password 'economy' without creating a schema:

```
CREATE IDENT audit AS PROGRAM USING 'economy' WITHOUT SCHEMA;
```

Create a group ident:

```
CREATE IDENT mimer_admin_group AS GROUP;
```

Create a schema called MIMER_STORE:

```
CREATE SCHEMA mimer_store;
```

Create table CURRENCIES in the schema MIMER_STORE:

```
CREATE TABLE mimer_store.currencies (
      code CHARACTER(3) PRIMARY KEY,
      ...
```

Create schema called MIMER_STORE_NEW that contains sequence Z:

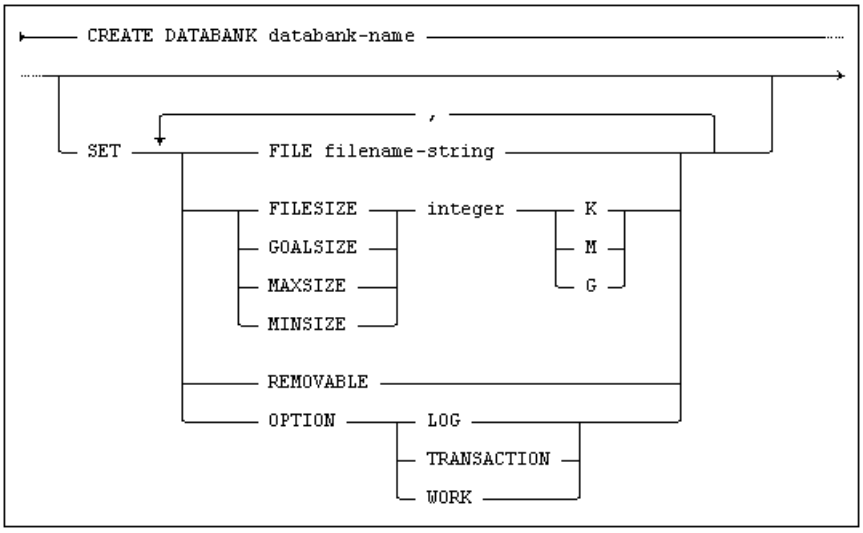
```
CREATE SCHEMA mimer_store_new
   CREATE SEQUENCE z;
```

Creating Databanks

A databank is the file where tables and sequences are stored. A Mimer SQL database may contain any number of databanks.

Create Databank Statement

The statement for creating a databank has the general form:



- The `CREATE DATABANK` clause defines the databank name.
- The optional `FILESIZE` clause is used to specify the initial file size (it will be dynamically extended as space is required). If the clause is omitted, an initial file size of 2000 kB is assumed. The optional `GOALSIZE`, `MAXSIZE` and `MINSIZE` attributes are used to manage the file size, see *Mimer SQL Reference Manual, Chapter 12, CREATE DATABANK*.
- The optional `FILE` clause defines the file where the databank is to be stored (the form of the filename follows the operating system file naming conventions). If the `FILE` clause is omitted, the file is created in the database home directory with the same name as `databank-name`.
- The optional `OPTION` clause defines the transaction handling and logging option, see *Logging Transactions* on page *92*. If the `OPTION` clause is omitted, the `TRANSACTION` option is assumed.

Examples

Create a databank called mimer_blobs with the default parameters:

Note: The default parameters are with TRANSACTION option and size 2000 kB. This databank is created in a file called “mimer_blobs.dbf”

```
CREATE DATABANK mimer_blobs;
```

Create the mimer_store databank with LOG option, allocate 1200 MB for it, and store it in a file called 'mstore.dbf':

```
CREATE DATABANK mimer_store SET FILESIZE 1200 M,
                                FILE 'mstore.dbf',
                                OPTION LOG;
```

At this point, the databank is empty.

Creating Tables

After the physical file space has been allocated on a disk for the databank, (CREATE DATABANK), you can create the tables. The basic CREATE TABLE statement defines the columns in the table, the primary key, any unique or foreign keys and which databank the table is to be stored in. Table names and column names may be up to 128 characters long.

As a convention, we have defined primary key column(s) as the first column(s) in the example definitions. However, this is not a necessity; primary key columns may be defined anywhere in the column list. Primary keys are always NOT NULL, so there is no need to explicitly state that in the table definition.

Create Table Statement

Example

Create the table CURRENCIES with three columns in the MIMER_STORE schema.

The table shall be as follows:

- Name the first column CODE, make it of the CHARACTER data type with a maximum of three characters.
- Name the second column CURRENCY, make it of the CHARACTER data type with a maximum of 32 characters and don't allow null values to be stored in the column.
- Name the third column EXCHANGE_RATE and make it of the data type DECIMAL with a total of twelve digits, four of which can be decimal values.
- Declare the CODE column as the primary key and place this table in the MIMER_STORE databank.

```
CREATE TABLE mimer_store.currencies (
      code CHARACTER(3),
      currency CHARACTER(32) NOT NULL,
      exchange_rate DECIMAL(12, 4),
      PRIMARY KEY(code))
   IN mimer_store;
```

The CREATE TABLE clause defines the name of the table followed by a column list, which includes the names of the columns in the table, their data type, if they should allow the null indicator and the primary key declaration. Each item in the column-list is separated from the next by a comma, and the entire list is enclosed in parentheses.

A table definition may only include one primary key clause. The primary key can be made up of more than one column.

The IN clause states which databank the table is to be stored in. This clause may be omitted; if the IN clause is not specified, Mimer SQL will select the 'best' databank in which to place the table.

The empty table now exists in the databank. Data is inserted into the table with the INSERT statement, see *Inserting Data* on page *85*.

The preceding example shows the simplest form of column list. The following variants may also be used:

- columns belonging to domains
- columns defined with collations
- default values (overriding any domain default for the column)
- columns not belonging to the primary key defined as NOT NULL
- unique constraints (in addition to the primary key)
- foreign key constraints
- check constraints.

The ITEMS Table

The ITEMS table in the example database is defined with many of the options that can be used in creating tables. See the *Mimer SQL Reference Manual, Chapter 12, CREATE TABLE*, for more information.

The ITEMS table is defined as follows:

```
CREATE TABLE items (
      item_id internal_id DEFAULT NEXT VALUE FOR item_id_seq,
      product_id internal_id CONSTRAINT itm_product_id_not_null NOT NULL,
      format_id format_id CONSTRAINT itm_format_id_not_null NOT NULL,
      release_date DATE,
      status CHAR DEFAULT 'A' CONSTRAINT itm_status_not_null NOT NULL
                              CONSTRAINT itm_status_valid
                                 -- Available, Deleted
                                 CHECK (status IN ('A', 'X')),
      price euros CONSTRAINT itm_price_valid
                     CHECK (price >= 4.99 AND price <= 366.00),
      stock SMALLINT CONSTRAINT itm_stock_not_null NOT NULL
                     CONSTRAINT itm_stock_valid CHECK (stock >= 0),
      reorder_level SMALLINT CONSTRAINT itm_reorder_level_not_null NOT NULL,
      ean_code BIGINT CONSTRAINT itm_ean_code_not_null NOT NULL,
      producer_id internal_id DEFAULT NULL,
      image_id internal_id DEFAULT NULL,
      CONSTRAINT itm_primary_key PRIMARY KEY(item_id),
      CONSTRAINT itm_ean_code_exists UNIQUE (ean_code),
      CONSTRAINT itm_products
         FOREIGN KEY (product_id) REFERENCES products(product_id)
         ON DELETE CASCADE ON UPDATE NO ACTION,
      CONSTRAINT itm_formats FOREIGN KEY (format_id) REFERENCES formats
         ON DELETE CASCADE ON UPDATE NO ACTION,
     CONSTRAINT itm_producers FOREIGN KEY (producer_id) REFERENCES producers
         ON DELETE NO ACTION ON UPDATE NO ACTION,
      CONSTRAINT itm_images FOREIGN KEY (image_id) REFERENCES images
         ON DELETE SET DEFAULT ON UPDATE NO ACTION) IN mimer_store;
```

The ordering of column specifications, key clauses and check conditions is not fixed. If desired, the key and check clauses can be written in association with the respective column specifications.

Each constraint is given a name which allows it to be dropped and modified separately. The constraint name is also useful if a program wants to find out which constraint failed for a particular statement.

Column Definitions

Columns should in general be defined as `NOT NULL` unless there is a specific reason for using the null value in the column (e.g. is the value not known, not applicable or given some other meaning). The presence of null values can often complicate the formulation of queries, see *Handling Null Values* on page *67*.

Note: Take particular care to exclude null from numerical columns which are to be used for mathematical operations.

Domains are used for many columns to help in maintaining database integrity. By using the same domain for columns in different tables, the column data types are guaranteed to be consistent. See *Creating Domains* on page *103* for more information

The Primary Key Constraint

The purpose of a primary key is to define a key value that uniquely identifies each table row, therefore the primary key value for each row in the table must be unique.

The primary key constraint can consist of more than one column in the table. The choice of columns to use as the primary key is determined by the relational model for the database, which is outside the scope of this manual.

Unique Constraints

A unique constraint can be defined for one or more columns in the table. The list of columns that make up the unique constraint are specified in the `UNIQUE` clause for the table when it is created.

Specifying `UNIQUE` in the definition of a column in the table is equivalent to supplying a list of one column in the `UNIQUE` clause for the table and effectively specifies a one-column unique constraint.

Foreign Keys – Referential Constraints

Use foreign keys to maintain integrity between the contents of related tables.

The effect of a foreign key is to constrain table data in a way that only allows a row in the referencing table which has a foreign key value that matches the specified key value of a row in the referenced table.

A referencing table row which has a foreign key value with the null value in at least one of the columns will always fulfil the referential constraint and therefore be acceptable as a row in the referencing table.

A foreign key constraint can be defined with a foreign key clause at `CREATE TABLE` or added afterwards using `ALTER TABLE`.

The table referenced in a foreign key clause can be an existing table or a table defined in the current statement (allowing self-referencing foreign keys at CREATE TABLE and circular foreign keys at CREATE SCHEMA).

The number of columns listed as FOREIGN KEY must be the same as the number of columns in the primary key or unique key of the REFERENCES table. See the CREATE TABLE syntax in the *Mimer SQL Reference Manual* for details.

The *n*th FOREIGN KEY column corresponds to the *n*th column in the primary key of the REFERENCES table, and the data types and lengths of corresponding columns must be identical.

A table definition may contain several FOREIGN KEY references. Each column in the table may be used in many FOREIGN KEY clauses, but only once per FOREIGN KEY clause.

Note: A table containing a foreign key reference or referenced in a foreign key must be stored in a databank with either the TRANSACTION or LOG option.

Foreign Key Example

The ITEMS table has four foreign key references:

```
CREATE TABLE items (
      item_id internal_id DEFAULT NEXT VALUE FOR item_id_seq,
      product_id internal_id CONSTRAINT itm_product_id_not_null NOT NULL,
      format_id format_id CONSTRAINT itm_format_id_not_null NOT NULL,
          .
          .
      producer_id internal_id DEFAULT NULL,
      image_id internal_id DEFAULT NULL,
          .
          .
      CONSTRAINT itm_products
         FOREIGN KEY (product_id) REFERENCES products(product_id)
         ON DELETE CASCADE ON UPDATE NO ACTION,
      CONSTRAINT itm_formats FOREIGN KEY (format_id) REFERENCES formats
         ON DELETE CASCADE ON UPDATE NO ACTION,
     CONSTRAINT itm_producers FOREIGN KEY (producer_id) REFERENCES producers
         ON DELETE NO ACTION ON UPDATE NO ACTION,
      CONSTRAINT itm_images FOREIGN KEY (image_id) REFERENCES images
         ON DELETE SET DEFAULT ON UPDATE NO ACTION)
          .
          .
```

These maintain referential integrity as follows:

- **FOREIGN KEY (product_id) REFERENCES products(product_id)**

  Data that is not present in the PRODUCT_ID column of the PRODUCTS table will not be accepted in the PRODUCT_ID column in the ITEMS table.

- **FOREIGN KEY (format_id) REFERENCES formats**

  Data that is not present in the FORMAT_ID column of the FORMATS table will not be accepted in the FORMAT_ID column in the ITEMS table.

- **FOREIGN KEY (producer_id) REFERENCES producers**

  Data that is not present in the PRODUCER_ID column of the PRODUCERS table will not be accepted in the PRODUCER_ID column in the ITEMS table.

- **FOREIGN KEY (image_id) REFERENCES images**

  Data that is not present in the IMAGE_ID column of the IMAGES table will not be accepted in the IMAGE_ID column in the ITEMS table.

Specifying ON DELETE

When defining a foreign key constraint it is possible to specify in an `ON DELETE` clause what action that shall take place if the corresponding record in the referenced table is deleted.

The possible actions are

- **NO ACTION**

  Any attempt to delete a key value that is referenced by a foreign key will fail. This action is the default behavior.

- **SET NULL**

  If a key value in the referenced table is deleted the corresponding values in the foreign key table is set to the null value

- **SET DEFAULT**

  If a key value in the referenced table is deleted the corresponding values in the foreign key table is set to the default value for the columns in the foreign key

- **CASCADE**

  If a key value in the referenced table is deleted the corresponding records in the foreign key table are also deleted

Check Constraints

Check constraints in table definitions are used to make sure that data in a column (or row) in the table fits certain conditions.

```
CREATE TABLE items (
          .
          .
      status CHAR (1) DEFAULT 'A' CONSTRAINT itm_status_not_null NOT NULL
                                  CONSTRAINT itm_status_valid
                                     -- Available, Deleted
                                     CHECK (status IN ('A', 'X')),
      price euros CONSTRAINT itm_price_valid
                     CHECK (price >= 4.99 AND price <= 366.00),
      stock SMALLINT CONSTRAINT itm_stock_not_null NOT NULL
                     CONSTRAINT itm_stock_valid CHECK (stock >= 0),
          .
          .
```

The check clause defined on the `PRICE` column extends any limitations imposed by the `EUROS` domain definition. The extension applies only to this table, and does not affect other columns in the database that belong to the `EUROS` domain:

```
CREATE DOMAIN euros AS NUMERIC(7, 2)
             CONSTRAINT euros_value_not_null CHECK (VALUE IS NOT NULL)
             CONSTRAINT euros_value_valid CHECK (VALUE > 0.0);
```

The constraint names, e.g. `ITM_PRICE_VALID` in the `ITEMS` table, can be used in an `ALTER TABLE` statement to drop the check constraint. All constraints, primary key, unique, not null and foreign key constraints can be named in this manner.

If no constraint name is given, a unique name is generated by the system. This name can be seen by using the describe statement in BSQL. See *Chapter 9, Mimer BSQL*.

Ensure that either the customer's e-mail address and password are both defined or that neither is defined:

```
CREATE TABLE customers (
          .
          .
     email VARCHAR(128) COLLATE english CHECK (char_length(trim(email)) > 0),
      password VARCHAR(18) CHECK (char_length(trim(password)) > 0),
          .
          .
      CONSTRAINT cst_email_password_cross_check
         CHECK (   (email IS NULL AND password IS NULL)
                OR (email IS NOT NULL AND password IS NOT NULL))
          .
          .
```

Creating Sequences

A sequence can be used to provide the default value for a table column or a domain, etc.

A sequence returns a series of integer values which is defined by a start value, a minimum value, a maximum value, an increment value, and whether the sequence is to be cyclic or not.

A sequence that has been initialized has a current value, which is returned from the function `CURRENT VALUE`. The function `NEXT VALUE` is used to initialize a sequence and to subsequently advance the current value of the sequence through its defined series of values.

A no cycle sequence will never return the same value twice.

Examples of Sequences

Create a sequence that returns odd numbers:

```
CREATE SEQUENCE seq_1 START WITH 1 INCREMENT BY 2 IN userdb;
```

Create a sequence that defines the following series of values: 1, 4, 7, 10:

```
CREATE SEQUENCE seq_2
   START WITH 1
   INCREMENT 3
   MAXVALUE 10
   NO CYCLE
IN DATABANK userdb;
```

Create a table that uses a sequence to set a column default value:

```
CREATE TABLE objinfo (objid INTEGER DEFAULT NEXT VALUE FOR obj_seq,
                      description NCHAR VARYING(1000));
```

Creating Domains

Domains are used as data types in column definitions when creating tables in order to:

- assist in keeping the database consistent
- validate the data (particular values or data type) accepted in the columns
- define default values for columns.

Create Domain Statement

The statement for creating domains has the general form:

```
CREATE DOMAIN domain-name
   AS data-type
   [DEFAULT default-value]
   [{[CONSTRAINT constraint_name] CHECK (check-condition)}...];
```

- The `CREATE DOMAIN` clause defines the domain name.
- The `AS` clause defines the domain data type.
- The `DEFAULT` clause defines a default value for the domain
- The `CHECK` clause defines the domain limits.

It is a good practice for maintaining the integrity of the database to define domains for as many columns as possible.

Domains with a Default Value

The default clause defines values that are inserted into the column when an explicit value is not specified or the keyword `DEFAULT` is used in an `INSERT` statement.

Examples

Define the default value '000000' for the domain SIXDIGITS:

```
CREATE DOMAIN sixdigits AS CHAR(6) DEFAULT '000000';
```

Define the session user's name as the default value for the domain USER_NAME:

```
CREATE DOMAIN user_name AS NVARCHAR(128) COLLATE SQL_IDENTIFIER
   DEFAULT SESSION_USER;
```

Domains with a Check Clause

Domains defining default values can also include check clauses. You could define the `SOUNDEX` domain as:

```
CREATE DOMAIN sixdigits AS CHAR(6) DEFAULT '000000'
                           CHECK (VALUE IS NOT NULL)
                           CHECK (CHAR_LENGTH(TRIM(VALUE)) = 6
                                  AND VALUE BETWEEN '000000' AND '999999');
```

This means that the null indicator will not be accepted into columns belonging to this domain and that the value must be a character string of six digits.

If the default value is defined as being outside the check constraint this ensures that an explicit value must always be inserted into the column.

Searching and Check Clauses

Specification of a `CHECK` clause means that only values for which the specified search condition evaluates to true may be assigned to a column belonging to the domain.

The search condition, see the *Mimer SQL Reference Manual, Chapter 10, Search Conditions*, in the `CHECK` clause may only reference the domain values (by using the keyword `VALUE`), constants, or the keywords `CURRENT_USER`, `SESSION_USER` and `NULL`.

Creating Functions, Procedures, Triggers and Modules

Functions and procedures are SQL routines that are stored in the data dictionary.

A module is a collection of SQL routines.

Triggers contain the same constructs as routines but are created on tables or views (depending on the type of trigger) and execute before, after or instead of a specified data manipulation operation.

Refer to the *Mimer SQL Reference Manual, Chapter 12, SQL Statements* for the syntax definitions for `CREATE FUNCTION`, `CREATE MODULE`, `CREATE PROCEDURE` and `CREATE TRIGGER`, and the *Mimer SQL Programmer's Manual, Chapter 11, Mimer SQL Stored Procedures* for a general discussion of the stored procedure functionality in Mimer SQL.

Creating Functions and Procedures

The `CREATE FUNCTION` statement is used to create a function that does not belong to a module and the `CREATE PROCEDURE` statement is used to create a procedure that does not belong to a module.

The format of the routine definition is the same in the `CREATE FUNCTION` and `CREATE PROCEDURE` statements as it is in a function or procedure declaration in a module.

Creating a Module

A module is created by using the `CREATE MODULE` statement and all the routines that belong to the module are defined by declaring them within the `CREATE MODULE` statement.

Routines cannot be added to a module after the module has been created and a routine cannot be removed from the module it belongs to. The routines in a module behave in all respects as single objects (e.g. `EXECUTE` privilege is applied on individual routines in a module, not the module). If the module is dropped, all the routines in it are dropped.

Creating a Trigger

The `CREATE TRIGGER` statement is used to define a trigger on a table or view.

Examples

Note: The examples that follow show the '`@`' character which is used in Mimer BSQL to delimit SQL statements whose syntax involves use of the normal end-of-statement character '`;`' before the actual end of the statement.

This is the case for many of the SQL/PSM statements. See *Chapter 9, Mimer BSQL* for details.

The '`@`' character may be used to delimit any statement. This is useful when dealing with large statement as the error reporting facility in BSQL shows more information in such cases.

Create a standalone function FUNC_1 with one input parameter of data type VARCHAR(20) that returns a value of data type INTEGER:

```
@
CREATE FUNCTION func_1(p1 VARCHAR(20)) RETURNS INTEGER
BEGIN
   ...
END
@
```

Create a standalone procedure PROC_1 with one input parameter of data type INTEGER and one output parameter of VARCHAR(20):

```
@
CREATE PROCEDURE proc_1(IN p_value1 INTEGER,
                        OUT p_value2 VARCHAR(20))
BEGIN
   ...
END
@
```

Create a module M1 containing 2 procedures, PROC_1 (with no parameters), PROC_2 (one input parameter, X, of data type INTEGER) and 1 function, FUNC_1 (with no parameters, returning an INTEGER):

```
@
CREATE MODULE m1
   DECLARE PROCEDURE proc_1()
   READS SQL DATA
   BEGIN
      ...
   END;

   DECLARE PROCEDURE proc_2(IN p_x INTEGER)
   MODIFIES SQL DATA
   BEGIN
      ...
   END;

   DECLARE FUNCTION func_1() RETURNS INTEGER
   READS SQL DATA
   BEGIN
      ...
   END;
END MODULE
@
```

Create a trigger that will execute after INSERT operations on table PRODUCTS:

```
@
CREATE TRIGGER products_after_insert AFTER INSERT ON products
REFERENCING NEW TABLE AS pdt
FOR EACH STATEMENT
BEGIN ATOMIC
   ...
END
@
```

Note: It is recommended that all functions, procedures and triggers are created by executing a command file so that they may be easily re-created in the event of being unintentionally dropped because of `CASCADE` effects following a drop.

The effect of `CASCADE` can be quite far-reaching where routines and modules are concerned, see the *Mimer SQL Programmer's Manual*.

The use of a command file also facilitates module re-definition by dropping an existing module, altering the CREATE MODULE statement in the command file and creating the new, redefined module.

Creating Views

A view is a logical subset of one or more base tables or views where columns are chosen by naming them and rows are chosen through specified conditions relating to column values.

Views are created, for example, so that users who need not see all the data in a single table are shown only the parts of the table that interest them (restriction views). Views can also be created as a combination of a number of columns from several different tables (join views).

Operations on views are actually performed on the underlying base tables. Certain view definitions do not allow data to be changed in the view (read-only views). See *Updatable Views* on page *90* for further details.

View names can be up to 128 characters long. Views are defined in terms of a SELECT statement; the result of the SELECT statement forms the contents of the view. There are no restrictions on which select statements that can be used in a view definition.

Creating a View

Create a restriction view on the CUSTOMERS table called CUSTOMER_DETAILS containing limited information:

```
CREATE VIEW customer_details
   AS SELECT surname, forename, address_1, address_2, town, postcode,
             title, date_of_birth, country_code, customer_id
   FROM customers;
```

In this case the columns in the view are named after the columns listed in the SELECT clause, since the view definition does not include a list of column names.

Create a join view, including an outer join:

```
CREATE VIEW product_details
   AS SELECT product, COALESCE(producer, ' ') AS producer, format,
             price, stock, reorder_level, release_date, ean_code,
             status, product_search, item_id, category_id, product_id,
             display_order, image_id
   FROM products JOIN items ON products.product_id = items.product_id
     JOIN formats ON items.format_id = formats.format_id
     LEFT OUTER JOIN producers ON items.producer_id = producers.producer_id;
```

Check Option

The check option can be used in updatable view definitions to limit the data that can be inserted into the view. If a check option is specified, data which does not fulfill the definition of the view cannot be inserted into the view.

```
CREATE VIEW swedish_customers
   AS SELECT *
   FROM customer_details
   WHERE country_code = 'SE'
   WITH CHECK OPTION;
```

The check option in the view definition (WITH CHECK OPTION) means that the COUNTRY_CODE column must be set to SE if new rows are inserted into the view or rows are updated using the view.

If there is an instead of trigger defined for the view, the WITH CHECK OPTION does not have any effect.

Creating Views Based on Other Views

Views can be based on other views. When a view is created based upon another view or views, the original view's limitations are carried over to the new view.

```
CREATE VIEW customer_addresses (surname, forename, recipient,
      address_1, address_2, town, postcode, country,
      salutation, customer_id)
   AS SELECT surname, forename,
      UPPER(recipient(title, forename, surname)), address_1,
      COALESCE(address_2, ' '), UPPER(town), UPPER(postcode), UPPER(country),
      salutation(title, forename, surname, date_of_birth, country_code),
      customer_id
   FROM customer_details
   JOIN countries ON code = country_code;
```

Creating Secondary Indexes

A secondary index is automatically used during searching when it improves the efficiency of the search.

Secondary indexes are maintained by the system and are invisible to the user.

Any column(s) may be specified as a secondary index, except columns declared using a LOB data type.

Columns in the PRIMARY KEY, the columns of a FOREIGN KEY and columns defined as UNIQUE are automatically indexed, (in the order in which they are defined in the key), and therefore creation of an index on these columns will not improve performance.

Secondary index tables are purely for Mimer SQL's internal use – you create the index, and Mimer SQL handles the rest.

If, for instance, you want to know which products were released on a specific date, Mimer SQL would have to search successively through the entire ITEMS table to find all items that matched the date you specified. If, however, you create a secondary index on release date, Mimer SQL would locate that date directly in the secondary index, which would save time.

Secondary indexes can improve the efficiency of data retrieval; but introduces an overhead for write operations (UPDATE, INSERT, DELETE). In general, you should create indexes only for columns that are frequently searched.

Indexes cannot be created directly on columns in views. However, since searching in a view is actually implemented as searching in the base table, an index on the base table will also be used in view operations.

Examples of Secondary Index

Create a secondary index called ITM_RELEASE_DATE on the RELEASE_DATE column in the ITEMS table:

```
CREATE INDEX itm_release_date ON items(release_date);
```

Primary key columns may also be included in a secondary index. If a table has the primary key columns A, B and C, the primary index would cover all three columns of the primary key.

The following combinations of the columns in the primary key are automatically indexed: A, AB and ABC. In addition, you could create secondary indexes on columns B, C, BC, AC etc.

An index may also be defined as UNIQUE, which means that the index value may only occur once in the table. (For this purpose, null is treated as equal to null). However, it is preferable to use unique constraints.

Create a UNIQUE secondary index called ITM_EAN_CODE on the EAN_CODE column in the ITEMS table:

```
CREATE UNIQUE INDEX itm_ean_code ON ITEMS(ean_code);
```

Sorting Indexes

The sorting order for indexes may be defined as ascending or descending. However, this makes no difference to the efficiency of the index, since Mimer SQL searches indexes forwards or backwards depending on the circumstances. I.e. the following two indexes are compatible, and only one of them is required.

```
CREATE INDEX idx_asc  ON t1 (c1 ASC)
CREATE INDEX idx_desc ON t1 (c1 DESC)
```

In some cases specifying the sort order makes sense. For example when ordering the result set by mixed orders, e.g:

```
SELECT * FROM t1
ORDER BY c1 ASC, c2 DESC;
```

In this case the index below is appropriate:

```
CREATE INDEX idx_mix ON t1 (c1 ASC, c2 DESC);
```

Creating Synonyms

Synonyms, or alternative names can be created for tables, views or other synonyms. You can create synonyms to personalize tables or just for your own convenience. Synonym names can be made up of a maximum of 128 characters.

Table names are 'qualified' by the name of the schema to which they belong. The qualified form of the table name is the schema name followed by the table name and the two are separated by a period.

Synonyms are particularly useful when several users refer to a common table, such as MIMER_STORE.ITEMS, MIMER_STORE.CURRENCIES, etc. With synonyms, several users can work in the same apparent environment without needing to refer to the tables by their qualified names.

Synonym Examples

The table ITEMS in the schema MIMER_STORE has the qualified name:

```
MIMER_STORE.ITEMS
```

The ident called MIMER_STORE need only refer to it as:

```
ITEMS
```

If other users wish to use this table, they must refer to it by its fully qualified name since they do not have the same name as the schema to which the table belongs.

If a user named JAMES, who wishes to refer to the ITEMS table, belonging to the schema MIMER_STORE, as simply ITEMS, he can create a synonym.

In the following example, the schema name JAMES is implied by default (which must also have been created by user JAMES if the CREATE is to succeed) because the synonym name is specified in its unqualified form (and the default schema name is the name of the current ident):

```
CREATE SYNONYM items FOR mimer_store.items;
```

Another user can then create his own synonym for the ITEMS synonym that now exists in schema 'JAMES', which has the fully qualified name:

```
JAMES.ITEMS
```

Commenting Objects

Comments may be stored against any of the following objects:

COLUMN	IDENT	PROCEDURE	SHADOW	TRIGGER
DATABANK	INDEX	SCHEMA	SYNONYM	TYPE
DOMAIN	METHOD	SEQUENCE	TABLE	VIEW
FUNCTION	MODULE			

Comments cannot be deleted – they can only be replaced by a new comment. A blank string may be provided as a comment if you want to suppress an existing comment.

Only the creator of the object may store a comment for it.

Comments are for information only and do not affect data retrieval or manipulation in any way.

Comments may be read with the DESCRIBE command, see *DESCRIBE* on page *133*, or by retrieving the appropriate columns from the INFORMATION_SCHEMA views, see the *Mimer SQL Reference Manual*.

Comment Example

Store the comment 'Holds currency details' on the CURRENCIES table:

```
COMMENT ON TABLE currencies IS 'Holds currency details';
```

Altering Databanks, Tables and Idents

The following sections explain how to alter databanks, tables and idents. you can also read about which objects you cannot alter.

Altering a Databank

Databanks can only be altered by their creator.

There are three uses for the `ALTER` statement:

- to change the physical file location for a databank
- to change the transaction and logging options on the databank
- to manage the file size allocated for the databank.

Examples

Change which file the MIMER_ORDERS databank is stored in from its previous file to file 'DISK2:MIMER_ORDERS.DBF':

Note: The file specification is in Alpha/OpenVMS format.

```
ALTER DATABANK mimer_orders SET FILE 'DISK2:[DBD]MIMER_ORDERS.DBF';
```

Note: This statement changes the filename stored for the databank in the data dictionary. It does not actually move the databank to the new location.

To move a databank, begin by copying or renaming the file in the operating system and then use `ALTER DATABANK... SET FILE` to change the file specification in the data dictionary.

Change the option on the MIMER_BLOBS databank from TRANSACTION to LOG:

```
ALTER DATABANK mimer_blobs SET OPTION LOG;
```

Set the size of the MIMER_BLOBS database to 2000 MB:

```
ALTER DATABANK mimer_blobs SET FILESIZE 2000 M;
```

Note: Use of the `ALTER DATABANK... SET FILESIZE` statement is not strictly necessary because databank files are extended dynamically. However, increasing the file allocation by a relatively large figure can help to minimize file fragmentation and improve response times.

Altering Tables

The `ALTER TABLE` statement changes the definition of the specified table and may only be used by the creator of the schema to which the table belongs.

There are the following uses for the `ALTER TABLE` statement:

- to add a new column or table constraint definition to an existing table
- to drop a column or table constraint from an existing table
- to change the default value for a column in an existing table

- to drop the default value for a column in a table
- to change a column in an existing table to have a specified data type, provided the old and new data types are assignment-compatible, see the *Mimer SQL Reference Manual* and the column is not referenced by any constraints or views

A new column created with the `ALTER TABLE... ADD` statement is appended to end of the existing column list. The new column will include the default value defined for the column or defined for the domain to which it belongs or, if no default value exists, the null value.

Note: If a column added to a table is defined as `NOT NULL`, then it must have a default value defined or belong to a domain which has a default value, because the `NOT NULL` column cannot be given null values.

Examples

Add a column called CREDIT_RATING with a data type of CHAR(1) to the CUSTOMERS table:

```
ALTER TABLE customers ADD credit_rating CHAR(1);
```

This creates a column containing the null value in each row in the table.

If a constraint is added to a table, the data in the table is checked to ensure it fulfills the restriction in the constraint.

Drop the column DATE_OF_BIRTH from the table CUSTOMERS, subject to the condition that there are no other objects dependent on this column:

```
ALTER TABLE customers DROP date_of_birth RESTRICT;
```

Drop the column DATE_OF_BIRTH from the table CUSTOMERS, if dependent objects exist, these are dropped as well:

```
ALTER TABLE customers DROP date_of_birth CASCADE;
```

Change the length of the column FORMAT in the table FORMATS:

```
ALTER TABLE formats ALTER COLUMN format VARCHAR(32);
```

Change the default value for the column REORDER_LEVEL, the new default value is one:

```
ALTER TABLE items ALTER reorder_level SET DEFAULT 1;
```

Drop the check constraint ITM_PRICE_ILLEGAL from the ITEMS table:

```
ALTER TABLE items DROP CONSTRAINT itm_price_valid;
```

Redefine a foreign key constraint for the CUSTOMERS table:

```
ALTER TABLE customers DROP CONSTRAINT cst_countries;
ALTER TABLE customers ADD CONSTRAINT cst_countries
   FOREIGN KEY (country_code) REFERENCES countries
      ON DELETE CASCADE ON UPDATE NO ACTION;
```

Drop the default value for the column REGISTERED:

```
ALTER TABLE customers ALTER registered DROP DEFAULT;
```

Note on Dropping

When dropping a column from a table, the CASCADE and RESTRICT keywords can be used to specify the action that will be taken on objects that are dependent on the dropped column.

If CASCADE is specified, dependent objects are also dropped. For instance if a dropped column is part of a primary key, the primary key will also be dropped.

If RESTRICT (the default) is specified and there are other objects affected, the statement will be aborted, with an error condition. See also, *Dropping Objects from the Database* on page *113*.

Altering Idents

Only passwords can be altered with the ALTER IDENT statement. Ident names cannot be altered.

USER and PROGRAM idents can change their own password if they so wish.

Passwords can also be changed by the creator of the ident. Also, an ident without a password is not allowed to set the password, only the creator of the ident may do this.

Change the ident MIMER_ADM's password to 'evjkl9u'.

```
ALTER IDENT mimer_adm SET PASSWORD 'evjkl9u';
```

Objects Which May Not Be Altered

Domains, functions, procedures, modules, triggers, views and indexes cannot be altered. It is therefore important that you think through your domains and views thoroughly and carefully before you create them to make sure that they suit the needs of your database.

The functions and procedures contained in a module are created when the module is created and thereafter no alterations can be made to the module (the module and all the routines contained in it can, of course, be dropped).

The next section will discuss dropping objects and the results of this on the database.

Dropping Objects from the Database

The DROP statement is used to drop the following objects from the database:

COLLATION	INDEX	SEQUENCE	TABLE
DATABANK	METHOD	SHADOW	TRIGGER
DOMAIN	MODULE	STATEMENT	TYPE
FUNCTION	PROCEDURE	SYNONYM	VIEW
IDENT	SCHEMA		

The CASCADE or RESTRICT keywords may be used to specify the action to be taken if other objects exist that are dependent on the object being dropped:

- If RESTRICT (the default) is specified, an error is returned if other objects are affected and the drop operation is aborted.
- If CASCADE is specified, dependent objects are dropped as well.

System database objects can only be dropped by their creator. Private database objects can only be dropped by the creator of the schema to which they belong.

Therefore use caution when using the DROP statement with CASCADE, as the operation may have a recursive effect on all objects relating to it. For example, when a table is dropped, all views, synonyms, routines and triggers created on or referencing that table are also dropped.

The DROP statement removes whole objects from the database. It cannot be used to remove columns from tables, this is done by the ALTER TABLE statement, see *Altering Tables* on page *111*.

Dropping Databanks and Tables

Drop the CURRENCIES table:

```
DROP TABLE currencies RESTRICT;
```

If the keyword CASCADE is specified, all views, synonyms and indexes based on CURRENCIES are also dropped as well as any functions, procedures and triggers referencing the table.

Drop the MIMER_STORE databank:

```
DROP DATABANK mimer_store RESTRICT;
```

If the keyword CASCADE is specified, all tables in the MIMER_STORE databank are also dropped and any views, synonyms, triggers and indexes based on those tables are also dropped as well as any functions, procedures and triggers referencing any of the dropped objects.

An attempt is automatically made to delete the physical databank file when a databank is dropped.

There may be occasions, because of access rights issues in the file system, when the database server's attempt to delete the physical databank file might fail. If recommended procedures for databank file management are followed, see the *Mimer SQL System Management Handbook*, the databank file should be deleted correctly.

Dropping Sequences

When a sequence is dropped, all the objects (i.e. constraints, domains, functions, procedures, default values, triggers and views) referencing the sequence are also dropped.

Drop the CUSTOMER_ID_SEQ sequence:

```
DROP SEQUENCE customer_id_seq CASCADE;
```

The specification of CASCADE ensures that the sequence is dropped even if it is being referenced by other objects in the database.

Dropping Domains

When a domain is dropped, columns using the domain retain the properties of the domain through the creation of column constraints.

Drop the EUROS domain:

```
DROP DOMAIN euros CASCADE;
```

Note: If you re-create a domain that has been dropped, the domain will be seen as a completely new domain and it will not be associated with any columns that belonged to the old domain.

To change the restrictions on the columns that were defined with a domain that has been dropped, use the `ALTER TABLE` statement.

Dropping Idents

When an ident is dropped, everything that the ident has created (including other idents and everything created by those idents) as well as all privileges granted by the ident are dropped. For this reason, physical users should never own objects, except for synonyms and personal views.

Drop the MIMER_ADM ident:

```
DROP IDENT mimer_adm RESTRICT;
```

Dropping Functions, Modules, Procedures and Triggers

The effect of using the keyword `CASCADE` can be rather dramatic when modules, routines and triggers are dropped. For this reason it is recommended that all those objects are created by running a command file so they can be easily reconstructed in case of being dropped by mistake.

Drop the function called MIMER_STORE_BOOK.FORMAT_ISBN:

```
DROP FUNCTION mimer_store_book.format_isbn CASCADE;
```

Drop the procedure called COMING_SOON:

```
DROP PROCEDURE coming_soon CASCADE;
```

Drop the module called MIMER_STORE_MUSIC.ROUTINES:

```
DROP MODULE mimer_store_music.routines CASCADE;
```

Drop the trigger called PRODUCTS_AFTER_INSERT:

```
DROP TRIGGER products_after_insert CASCADE;
```

About Dropping Modules and Routines

The following points should be noted when dropping modules and routines:

- When a module is dropped, all the routines contained in it will be dropped.
- If a routine is dropped and it is referenced from another object, the referencing object will also be dropped.
- If a routine belonging to a module is to be dropped as a consequence of a cascade, only that routine is dropped (the other routines in the module and the module itself will remain unaffected).

Chapter 8

Defining Privileges

Privileges control the operations which users are allowed to perform in the database. Well-structured privileges are essential for maintaining data security.

There are three types of privileges:

- System privileges, which give the right to create global objects within the database.
- Object privileges, which give rights over certain specified objects in the database.
- Access privileges, which give rights of access to the data in a specified table or view.

System privileges are granted to the system administrator upon installation, and may be passed on to other idents. Objects and access privileges are initially granted only to the creator of an object. The creator may however pass the privileges on to other idents.

Granting and Revoking Privileges

Privileges are granted to idents with the `GRANT` statement and revoked from idents with the `REVOKE` statement.

All privileges may be granted with the 'with grant option', which means that the receiver of the privilege in turn has the right to grant that privilege to other idents.

The creator of an object is automatically granted full privileges on that object with grant option. Thus the creator of:

- a group is automatically a member of that group
- a program ident may enter it
- a table has full access privileges
- a schema may create objects in it and drop them, etc.

When privileges that were granted with the 'with grant option' are revoked, the right to grant those privileges to other idents is also revoked.

The 'with grant option' can be revoked separately without revoking the privilege itself.

Idents may only grant privileges that they themselves possess to other idents, that is, idents cannot grant privileges to themselves.

Likewise, privileges may only be revoked by the grantor - idents cannot revoke privileges from themselves.

Certain operations are not controlled by explicit privileges, but may only be performed by the creator of the object involved. These operations include `ALTER` (with the exception of `ALTER IDENT`, which may be performed by either the ident himself or by the creator of the ident), `DROP` and `COMMENT`.

Ident Structure

In the initial installation, one user ident, the system administrator with user ident name SYSADM, is automatically created.

SYSADM Privileges

The system administrator has the following privileges (with grant option):

- BACKUP
- DATABANK
- IDENT
- SCHEMA
- SHADOW
- SELECT access on all tables and views in the data dictionary.

The system administrator is ultimately responsible for the structure of the whole system. In other respects, however, the system administrator is an ordinary user ident in the system.

There is no ident in Mimer SQL with automatic right of access to all objects within the system.

It is quite possible, and may be advisable especially in large system, that the system administrator is prevented from accessing the actual contents of the database; the system administrator's job is to manage objects in the system, not work on the data.

About System Utilities

Certain system utilities may only be run by idents with BACKUP or SHADOW privilege, see the *Mimer SQL System Management Handbook*.

When granting privileges, the keyword PUBLIC refers to a logical group that covers all idents in the database, including those created in the future.

Recommendations for Ident Structure

The following general recommendations can be made for structuring the idents in a system:

- Functional roles within the system, generally defined by one or more applications that are run, should be assigned to program idents. These are not coupled to any physical individual or group of individuals and thus have a lifetime independent of turnover of personnel.

  The system administrator is just such a function, but is coupled to a user ident rather than a program ident for practical purposes.

- People accessing the system are represented by USER idents. They may be dropped if the person concerned leaves the company.
  User idents should not be granted privileges directly, other than membership in groups. A USER ident with an OS_USER login is allowed access to the database on the authorization of a valid log-in to the operating system.

- Group idents are used to represent logical users of the system. Privileges are granted to groups rather than to individual programs or users. The individual idents are granted membership in the group to which they belong, and thereby gain the correct access to the system.
- `USER` idents should not in general be able to create objects. This is performed by specifying `WITHOUT SCHEMA` when the user is created. In this way, individual user idents may be dropped with no cascading effects.
- `WITH GRANT OPTION` should be used sparingly and the ident hierarchy kept shallow. This minimizes the chance of undesired cascading revocation of privileges.

If these recommendations are followed, maintenance of the ident structure in the system is simplified. Access to the contents of the database is granted to relatively few group idents instead of many individual programs or users, and when a physical individual leaves the company, their user ident can be dropped with no cascading consequences.

Granting Privileges

The following sections explain how to grant system, object and access privileges.

Granting System Privileges

System privileges are granted to the system administrator at the time of installation of the system. System privileges refer to global information, that affects the database as a whole.

The system privileges are:

Privilege	Explanation
`BACKUP`	The right to perform backup and restore operations.
`DATABANK`	The right to create databanks.
`IDENT`	The right to create idents and schemas.
`SCHEMA`	The right to create schemas.
`SHADOW`	The right to create shadows and perform shadow control operations.
`STATISTICS`	The right to execute the `UPDATE STATISTICS` statement.

Examples

Give the ident MIMER_STORE the privilege to create new databanks:

```
GRANT DATABANK TO mimer_store;
```

Give the ident MIMER_STORE the privilege to create new idents with grant option:

```
GRANT IDENT TO mimer_store WITH GRANT OPTION;
```

Granting Object Privileges

Object privileges are held by idents on database objects (functions, procedures, programs, groups, tables, domainssequences).

The object privileges are:

Privilege	Explanation
EXECUTE	The right to execute a function or procedure, or the right to enter a specified program ident.
MEMBER	Membership in a specified group ident.
TABLE	The right to create tables in a specified databank.
SEQUENCE	The right to create sequences in a specified databank.
USAGE	The right to specify the named domain where a data type would normally be specified (in contexts where use of domains are allowed), or the right to use a specified sequence.

Examples

Give MIMER_WEB the privilege to execute the COMING_SOON procedure:

```
GRANT EXECUTE ON PROCEDURE coming_soon TO mimer_web;
```

Make MIMER_ADM a member of the MIMER_ADMIN_GROUP group with grant option:

```
GRANT MEMBER ON mimer_admin_group TO mimer_adm WITH GRANT OPTION;
```

Give MIMER_ADM the privilege to create new tables in the MIMER_STORE databank:

```
GRANT TABLE ON mimer_store TO mimer_adm;
```

Give the members of the MIMER_ADMIN_GROUP group the privilege to use the NAME domain:

```
GRANT USAGE ON DOMAIN name TO mimer_admin_group;
```

Granting Access Privileges

Access privileges define what data the idents are allowed to manipulate in tables.

There are five access privileges:

Privilege	Explanation
SELECT	The right to read the table contents.
INSERT	The right to add new rows to the table (this privilege may be limited to specified columns within the table).
DELETE	The right to remove rows from the table.

Privilege	Explanation
UPDATE	The right to change the contents of existing rows in the table (this privilege may be limited to specified columns within the table).
REFERENCES	The right to use the primary or unique key of the table as a foreign key reference (this privilege may be limited to specified columns within the table).

The keyword ALL may be used as shorthand for all of privileges that the grantor holds with grant option.

Examples

Give the MIMER_ADMIN_GROUP group the privileges to read and update rows from the PRODUCERS table:

```
GRANT SELECT, UPDATE ON producers TO mimer_admin_group;
```

Give MIMER_USER_GROUP all privileges that you hold on the table COUNTRIES and give them the right to pass those privileges on:

```
GRANT ALL ON countries TO mimer_user_group WITH GRANT OPTION;
```

Give MIMER_ADMIN_GROUP the privilege to select all rows in the CURRENCIES table, with the privilege to only update the EXCHANGE_RATE column:

```
GRANT SELECT, UPDATE(exchange_rate) ON currencies TO
  mimer_admin_group;
```

Give everyone the privilege to select all rows in the CURRENCIES table:

```
GRANT SELECT ON currencies TO PUBLIC;
```

Give MIMER_ADM the right to use the ITEMS table as a foreign key:

```
GRANT REFERENCES ON mimer_store.items TO mimer_adm;
```

Revoking Privileges

Privileges can only be revoked by the grantor. Care must be taken when revoking privileges, especially when those privileges were granted 'with grant option'. Revoking such privileges from an ident can have recursive effects on all idents who have been granted privileges by that ident. See *Recursive Effects of Revoking Privileges* on page *123* for details.

The keywords CASCADE and RESTRICT can be used in the REVOKE statements to control whether the recursive effects should be allowed or not. If RESTRICT (the default) is specified and any recursive effects are identified the whole revoke operation will fail, leaving all objects intact. If the keyword CASCADE is specified, the revoke operation will proceed with recursive effects.

Privileges granted to a group cannot be revoked separately from individual members of the group. To revoke a group privilege from an individual, either revoke the privilege from the group or revoke the membership of the individual in the group.

If a privilege has been granted with the WITH GRANT OPTION it is possible to revoke the grant option only. That is, the ident looses the right to grant the privilege to other idents. The ident still has the privilege, but privileges granted to other idents are revoked.

Revoking System Privileges

Revoking system privileges does not affect objects already created under the authorization of the privilege.

The following examples show how to revoke system privileges.

Take away the privilege to create new databanks from the ident MIMER_STORE:

```
REVOKE DATABANK FROM mimer_store RESTRICT;
```

Take away the privilege to create new idents from the ident MIMER_STORE:

```
REVOKE IDENT FROM mimer_store RESTRICT;
```

Revoking Object Privileges

The following examples show how to revoke object privileges.

Take away the privilege to execute the COMING_SOON procedure from MIMER_WEB:

```
REVOKE EXECUTE ON PROCEDURE coming_soon FROM mimer_web;
```

Take away MIMER_ADM's membership of the MIMER_ADMIN_GROUP group:

```
REVOKE MEMBER ON mimer_admin_group FROM mimer_adm;
```

Take away the right to use the domain NAME from the group MIMER_ADMIN-_GROUP:

```
REVOKE USAGE ON DOMAIN name FROM mimer_admin_group;
```

Note: Revoking usage on domain prevents the ident from using that domain as a data type in new definitions, any existing definitions created by the ident will remain unaffected.

Revoking Access Privileges

The following examples show how to revoke access privileges.

Revoke the privilege to read and update rows from the PRODUCERS table from the group MIMER_STORE_GROUP:

```
REVOKE SELECT, UPDATE ON producers FROM mimer_admin_group;
```

When the REFERENCES privilege on a table is taken away from an ident, all foreign key links referencing that table are removed.

Revoke the right to use columns in the ITEMS table as a foreign key from MIMER_ADM:

```
REVOKE REFERENCES ON mimer_store.items FROM mimer_adm RESTRICT;
```

Revoke the right to grant select on the COUNTRIES table. Any grants that members of the group have made will also be revoked:

```
REVOKE GRANT OPTION FOR SELECT ON countries
   FROM mimer_user_group CASCADE;
```

The Keyword ALL

The keyword ALL may be used as a shorthand for all the privileges that may be revoked in the current context.

Recursive Effects of Revoking Privileges

If CASCADE is specified in a REVOKE statement, the following recursive effects may occur:

- If a privilege WITH GRANT OPTION is revoked from an ident, all instances of that privilege granted to other idents under the authorization of the WITH GRANT OPTION are also revoked. Privileges granted for procedures, functions and triggers that reference objects accessed by the WITH GRANT OPTION will also disappear.
- If SELECT privilege on a table is revoked from an ident, views created by the ident under the authorization of that SELECT privilege are dropped.
- If REFERENCE privilege on a table is revoked from an ident, any FOREIGN KEY constraints in tables created by that ident under the authorization of that REFERENCE privilege are removed.
- If the privilege held by an ident on an object referenced in a routine or trigger is revoked, the routine or trigger will be dropped. (This applies to EXECUTE on a routine, USAGE on a sequence or an access privilege on a table or view held WITH GRANT OPTION).

Dependencies

The recursive effect of revoking a privilege depends on how many instances of that privilege have been granted. An ident will hold more than one instance of a privilege when it has been granted more than once (by different idents, as an ident cannot grant the same privilege to the same ident more than once).

One or more of those instances may have been granted WITH GRANT OPTION.

The data dictionary keeps a record of which instance of a privilege has WITH GRANT OPTION and which does not.

The recursive effects will occur only when the last instance of the required privilege is revoked. That is, when the last instance of the privilege held WITH GRANT OPTION is revoked from an ident, all instances of the ident granting the privilege to others will be withdrawn; and when the last instance of the privilege is revoked from the ident, the cascade effects of the ident no longer holding the privilege will occur.

This is illustrated in the example cases that follow:

CASE 1

1. A grants with grant option to M
   M grants to X
2. B grants with grant option to M
   M grants to Y
3. A revokes from M
   Both X and Y keep privileges
4. B revokes from M
   Both X and Y lose privileges

CASE 2

1. A grants with grant option to M
2. B grants without grant option to M
   M grants to X
   M grants to Y
3. A revokes from M
   M loses grant option
   Both X and Y lose privileges
4. B revokes from M
   M loses privilege

As a consequence of the cascading effects of revoking privileges, careful advance planning of the hierarchical structure of idents in a system can be essential to the long term viability of the system.

An unplanned ident structure can easily become impossible to overview and control after a relatively short period of system use.

Chapter 9

Mimer BSQL

This chapter discusses Mimer BSQL, a command line oriented tool for executing SQL statements both in scripts and interactively.

Other SQL Tools

DbVisualizer, by DbVis Software (https://www.dbvis.com), is included in the Mimer SQL distribution. (Not for the OpenVMS platform.)

Running BSQL

BSQL can be run by a script or interactively. Interactive operation can be used to execute statements entered directly or read from sequential files.

About Complex SQL Statements – @

Use the `@` character to delimit a complex SQL statement where the normal end-of-statement character '`;`' appears before the end of the statement (e.g. when creating functions, procedures and triggers.)

Example

```
@
create function capitalize(str nvarchar(1000)) returns nvarchar(1000)
begin atomic
  declare outstr nvarchar(1000);
  declare strlen integer;
  declare n integer default 2;
  set strlen = length(str);
  set outstr = upper(substring(str from 1 for 1));
  while n <= strlen do
    set outstr = outstr || case when substring(str from n - 1 for 1) = ' ' then
                                    upper(substring(str from n for 1))
                                else
                                    lower(substring(str from n for 1))
                            end;
    set n = n + 1;
  end while;
  return outstr;
end
@
```

Running BSQL From a Script

Create a script file with the following contents:

- the command to start BSQL
- the username
- the password
- the SQL statements and BSQL commands
- the `EXIT` command (or end of file).

Unicode Pipe Support in Console Programs on Windows

All Mimer SQL console programs such as BSQL, can pipe Unicode files. The files can be any of the Unicode formats supported by Mimer SQL such as UTF16 big and little endian, and UTF8.

When output is piped to a file, the input decides the type of the output file. If the input file is ASCII, the output will also be ASCII. If the input file is UTF16, the output will also be UTF16. If the input is from the keyboard, the output will be an UTF16 file on Windows. For example:

```
BSQL < UNIFILEIN.TXT > UNIFILEOUT.TXT
```

Security and Script Jobs

For unattended operation, a script file must either include the Mimer SQL ident username and password in explicit form or connect using an `OS_USER` login.

For security reasons, make sure that your script files are well protected and/or remove your password from the file after execution.

Alternatively, SQL statements and BSQL commands may be written in a sequential file without username and password, and executed with the `READ` command from an interactive BSQL session.

Running BSQL

How you start BSQL depends on your operating system.

BSQL Command-line Arguments

The BSQL command line argument syntax is:

```
bsql [-m|-s] [-u user] [-p pass] [-q SQL [-c]] [database]

bsql [--multi|--single] [--user=user] [--password=pass]
    [--query=SQL [--continue]] [database]

bsql [-v|--version] | [-?|--help]
```

Options

Windows & Unix-style	VMS-style	Function
`-c` `--continue`	`/CONTINUE`	The switch can be used together with the `--query` switch to indicate that the BSQL program is not terminated after the execution of the query.
`-m` `--multi`	`/MULTI`	Connects to the database in multi-user mode.
`-p` *password* `--password=`*password*	`/PASSWORD=`*password*	Password for ident. If the switch is omitted the user is prompted for a password, unless OS_USER is specified with the username switch, as described above. **VMS:** Note that in an Open VMS environment it might be necessary to enclose the password in quotation marks as the value otherwise is translated to upper case.
`-q` *query* `--query=`*query*	`/QUERY=`*query*	`query` can be any BSQL command or SQL statement. If a query is supplied, BSQL will terminate immediately after the query has been processed.
`-s` `--single`	`/SINGLE`	Connects to the database in single-user mode.
`-u` *username* `--username=`*username*	`/USERNAME=`*username*	Ident name to be used in connect. If the switch is not given the user is prompted for a username. To connect using OS_USER, give `-u ""`, `--username=""`, or `/USERNAME=""`.
`-v` `--version`	`/VERSION`	Display version information.

Windows & Unix-style	VMS-style	Function
`-?` `--help`	`/HELP`	Show help text.
database	*database*	Specifies the name of the database to access. If a database name is not specified, the default database will be accessed, see *Mimer SQL System Management Handbook, Chapter 3, The Default Database*.

If a database name is not specified, the default database will be accessed.

If neither `--single` nor `--multi` is specified for the optional mode flag, the way the database is accessed will be determined by the setting of the `MIMER_MODE` variable, see *Mimer SQL System Management Handbook, Appendix A, Specifying Single-user Mode Access*.

If this is not set, it will be accessed in multi-user mode.

If multiple instances of a qualifier is given, the last one is used. For example

```
bsql --single --multi --username=t1 --username=t2
```

is valid and means connect user `t2` in multi-user mode.

Linux examples

Start BSQL and connect user cosmo with password Kramer:

```
bsql --username=cosmo --password=Kramer
```

Start BSQL, connect using OS_USER, execute a query, and then leave:

```
bsql --user="" --query="select * from \"SomeTable\" where user = 'COSMO'"
```

VMS examples

Start BSQL and connect user cosmo with password Kramer:

```
bsql /USERNAME="cosmo" /PASSWORD="Kramer"
```

Start BSQL, connect using OS_USER, execute a query, and then leave:

```
bsql /USERNAME="" /QUERY="select * from ""SomeTable"" where user = 'COSMO'"
```

Note: You can also use the Unix-style syntax in OpenVMS.

Windows examples

Start BSQL and connect user cosmo with password Kramer:

```
bsql --username=cosmo --password=Kramer
```

Start BSQL, connect using OS_USER, execute a query, and then leave:

```
bsql --user="" --query="select * from \"SomeTable\" where user = 'COSMO'"
```

To start Mimer BSQL from the Windows Start button:

Click **Start**, navigate to your **Mimer SQL** program group and select **Batch SQL** which is found in the Utilities sub-group.

Logging IN

Starting BSQL displays the following:

```
$ bsql
Mimer SQL Command Line Utility, version 11.0.7A
Copyright (C) Mimer Information Technology AB. All rights reserved.

Username:
```

After you have entered a username and a correct password, the BSQL prompt is displayed:

```
SQL>
```

You can now enter BSQL specific commands and general SQL statements.

BSQL Command Line Editing – Linux

Command line editing is available in the BSQL program, which uses a line-oriented interface.

The following functions are available:

Use:	To:
ctrl-a	Move to beginning of command
ctrl-b	Move backwards in command
ctrl-d	Delete current character
ctrl-e	Move to end of command
ctrl-f	Move forwards in command
ctrl-h	Delete previous character
ctrl-k	Delete after current position in command
ctrl-n	Next command
ctrl-o	Execute retrieved command and get next from history list
ctrl-p	Previous command
ctrl-r	Retrieve command by search condition
ctrl-t	Change place for the previous two characters
ctrl-u	Delete command
ctrl-w	Delete before current position in command
ctrl-<space>	Set mark in command (or 'esc <space>')
ctrl-x ctrl-x	Go to mark set by 'ctrl <space>'
ctrl-x ctrl-h	Show the history list

Use:	To:
ctrl-x ctrl-r	Retrieve command by history list number
esc h	Delete previous word
esc d	Delete next word
esc b	Move to previous word
esc f	Move to next word

You can use the arrow keys for command retrieval and for positioning the cursor within a line, i.e. the same function as for ctrl-b, ctrl-f, ctrl-n and ctrl-p.

To change the number of commands that can be held in the history list, the environment variable MIMER_HISTLINES can be used (the default is 23).

Note: The operating system may have control sequences set that, if they overlap, override those described above. E.g. the settings can be listed using the Linux stty -a command.

BSQL Commands

Command	Function
CLOSE	Closes active log files, see *CLOSE* on page *132*.
DESCRIBE	Describes a specified object, see *DESCRIBE* on page *133*.
EXIT	Leaves BSQL, see *EXIT* on page *142*.
GET DIAGNOSTICS	Presents status and diagnostics information, see *GET DIAGNOSTICS* on page *142*
LIST	Lists information on a specified object, see *LIST* on page *143*.
LOG	Logs input, output or both on a sequential file, see *LOG* on page *146*.
READ INPUT	Reads commands from a sequential file, see *READ INPUT* on page *147*.
READLOG	Obtains information about logged operations, see *READLOG* on page *147*.
SET ECHO	Specifies whether lines are echoed to the BSQL window during READ INPUT, see *SET ECHO* on page *151*.
SET EXECUTE	Activate or deactivate the execution of queries, see *SET EXECUTE* on page *152*.
SET EXPLAIN	Activate or deactivate the explain facility, see *SET EXPLAIN* on page *152*.

Command	Function
SET HEADER	Activate or deactivate display of column headings, see *SET HEADER* on page *153*.
SET LINECOUNT	Sets the BSQL page size, see *SET LINECOUNT* on page *154*.
SET LINESPACE	Sets the number of blank lines between each output record, see *SET LINESPACE* on page *154*.
SET LINEWIDTH	Sets the BSQL page width, see *SET LINEWIDTH* on page *155*.
SET LOG	Stops or resumes logging input, output or both, see *SET LOG* on page *155*.
SET MAX_BINARY_LENGTH	Specifies the maximum display length for binary columns, see *SET MAX_BINARY_LENGTH* on page *155*.
SET MAX_CHARACTER_LENGTH	Specifies the maximum display length for character columns, see *SET MAX_CHARACTER_LENGTH* on page *156*.
SET MESSAGE	Specifies whether messages are displayed, see *SET MESSAGE* on page *156*.
SET OUTPUT	Specifies whether output should be written, see *SET OUTPUT* on page *156*.
SET PAGELENGTH	Defines the page length of output file, see *SET PAGELENGTH* on page *157*.
SET PAGEWIDTH	Defines the page width of output file, see *SET PAGEWIDTH* on page *157*.
SET SILENCE	Specifies whether or not messages or column headers should be displayed, see *SET SILENCE* on page *157*.
SET STATISTICS	Specifies whether or not statement statistics should be displayed, see *SET STATISTICS* on page *158*.
SHOW SETTINGS	Displays current values of all set options, *SHOW SETTINGS* on page *158*.
TRANSACTIONS	Displays the menu for administration of distributed transactions, see *TRANSACTIONS* on page *159*.
WHENEVER	Sets action to be taken in response to an error or warning, see *WHENEVER* on page *160*.

BSQL commands are not case sensitive.

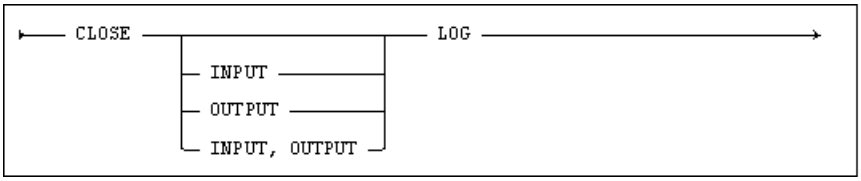
About BSQL Syntax Descriptions

For information on how to read the syntax diagrams that follow, please refer to *Mimer SQL Reference Manual, Chapter 2, Reading SQL Syntax Diagrams*.

CLOSE

Closes log files.

Syntax



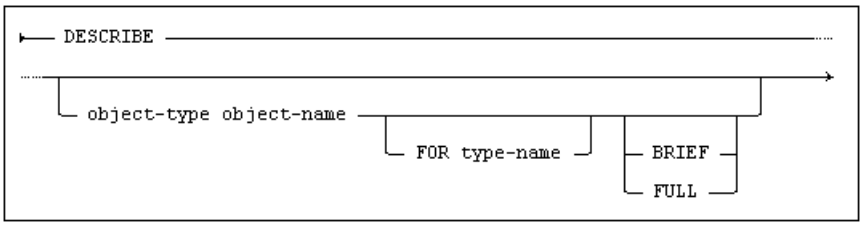
Description

The command closes the specified log file. If no log type is specified, all active log files are closed.

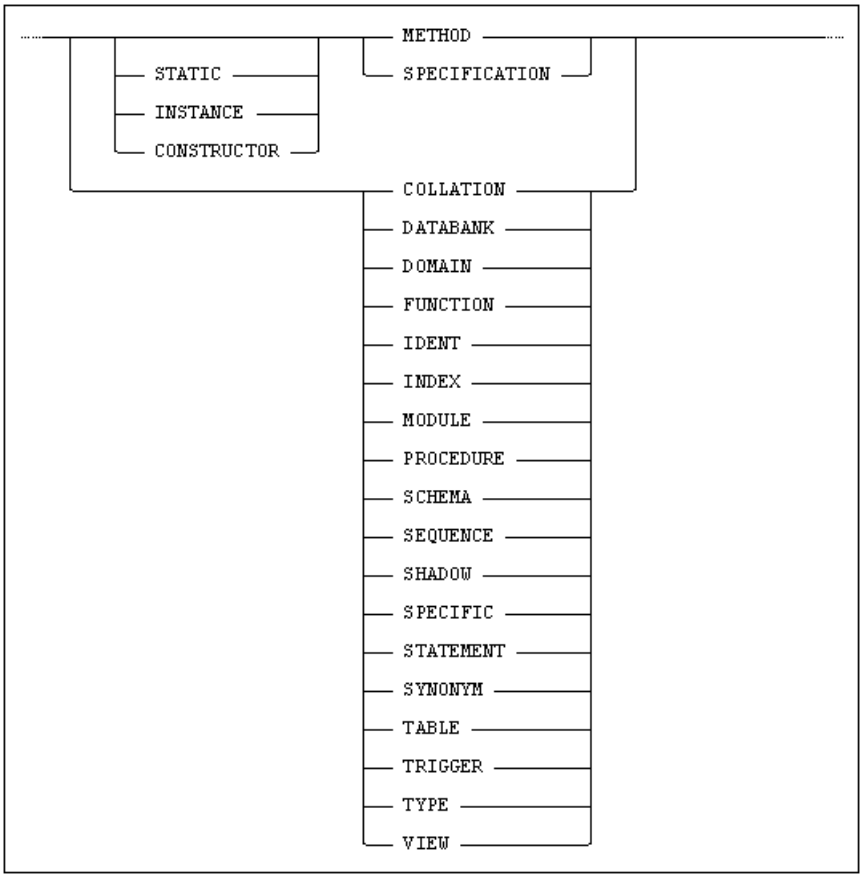
DESCRIBE

Describes a specified object.

Syntax



Where `object-type` is:



Description

The `DESCRIBE` command presents the following menu, when no object is specified:

```
                        Menu for describe

 1. Databank             8. View             15. Collation
 2. Domain               9. Module           16. Type
 3. Ident               10. Procedure        17. Method
 4. Index               11. Function         18. Specification
 5. Synonym             12. Trigger          19. Specific
 6. Table               13. Sequence         20. Statement
 7. View                14. Shadow            0. Exit
```

Choosing an item presents a submenu for choosing between different `DESCRIBE` functions – see the table that follows for details.

Entering an exclamation mark (`!`) in the Select field returns to the previous menu level. Entering a double exclamation mark (`!!`) terminates the `DESCRIBE` session.

Specifying an object type and name in the command executes the first menu choice for that object. If no object name is given, the user is prompted for a name.

Selection numbers should not be used in script files, since they may change in future versions.

Note: `DESCRIBE` is not available when connected to a Mimer SQL server of version 8.1 or older.

DESCRIBE DATABANK Options

`DESCRIBE DATABANK`	**Result**
`BRIEF`	Lists the following information on the specified databank: creator file space used allocated size filename databank option minsize goalsize maxsize removable option backup information tables sequences.
`BY TABLE PRIVILEGE`	Lists the following information on the specified databank: idents with table privilege.
`FULL`	Lists the following information on the specified databank: creator file space used allocated size filename databank option minsize goalsize maxsize removable option backup information tables sequences idents with table privilege comment creation date.

DESCRIBE DOMAIN Options

DESCRIBE DOMAIN	**Result**
BRIEF	Lists the following information on the specified domain: data type default value check constraints.
BY REFERENCES	Lists the following information on the specified domain: referenced objects referencing objects.
BY ACCESS	Lists the following information on the specified domain: idents with usage privilege.
FULL	Lists the following information on the specified domain: data type default value check constraints referenced objects referencing objects idents with usage privilege comment creation date.

DESCRIBE IDENT Options

DESCRIBE IDENT	**Result**
BRIEF	Lists the following information on the specified ident: creator ident type YES/NO if the ident has a schema with the same name or not YES/NO if the ident has a password or not A list of OS_USER logins defined for the ident privileges held by ident.
BY ACCESS	Lists the following information on the specified ident: accessible objects.
BY OWNERSHIP	Lists the following information on the specified ident: created objects.

DESCRIBE IDENT	**Result**
FULL	Lists the following information on the specified ident: creator ident type YES/NO if the ident has a schema with the same name or not YES/NO if the ident has a password or not A list of OS_USER logins defined for the ident accessible objects created objects comment creation date.

DESCRIBE INDEX Options

DESCRIBE INDEX	**Result**
BRIEF	Lists the following information on the specified index: table name and columns on which the index is defined sort order uniqueness index algorithm comment creation date.

DESCRIBE SYNONYM Options

DESCRIBE SYNONYM	**Result**
BRIEF	Lists the following information on the specified synonym: schema and name of referenced table/view comment creation date

DESCRIBE TABLE Options

DESCRIBE TABLE	**Result**
VERY BRIEF	Lists the following information on the specified table or view: column names and types.
BRIEF	Lists the following information on the specified table or view: column names and types default values constraints referenced domains indexes triggers.
BY ACCESS	Lists the following information on the specified table or view: idents with access.

DESCRIBE TABLE	**Result**
BY REFERENCES	Lists the following information on the specified table or view: referencing objects referenced objects.
FULL	Lists the following information on the specified table or view: column names and types default values constraints referencing objects referenced objects indexes triggers idents with access databank location comment creation date date when statistics were generated.

DESCRIBE VIEW Options

DESCRIBE VIEW	**Result**
BRIEF	Lists the following information on the specified view: view definition updatability check option comment creation date.

DESCRIBE MODULE Options

DESCRIBE MODULE	**Result**
BRIEF	List the following information on the specified module: module definition comment creation date.

DESCRIBE PROCEDURE Options

DESCRIBE PROCEDURE	**Result**
BRIEF	Lists the following information on the specified procedure: parameters return types access mode.
BY ACCESS	Lists the following information on the specified procedure: idents with execute privilege.

DESCRIBE PROCEDURE	**Result**
BY REFERENCES	Lists the following information on the specified procedure: referencing objects referenced objects.
FULL	Lists the following information on the specified procedure: parameters return types access mode idents with execute privilege referencing objects referenced objects source definition module name comment creation date.

DESCRIBE FUNCTION Options

DESCRIBE FUNCTION	**Result**
BRIEF	Lists the following information on the specified function: parameters return type access mode.
BY ACCESS	Lists the following information on the specified function: idents with execute privilege.
BY REFERENCES	Lists the following information on the specified function: referencing objects referenced objects.
FULL	Lists the following information on the specified function: parameters return type access mode idents with execute privilege referencing objects referenced objects source definition module name comment creation date.

DESCRIBE TRIGGER Options

DESCRIBE TRIGGER	Result
BRIEF	Lists the following information on the specified trigger: table name on which trigger is defined trigger event trigger type event time.
BY REFERENCES	Lists the following information on the specified trigger: referenced objects.
FULL	Lists the following information on the specified trigger: table name on which trigger is defined trigger event trigger type event time referenced objects source definition comment creation date.

DESCRIBE SEQUENCE Options

DESCRIBE SEQUENCE	Result
BRIEF	List the following information about the specified sequence: initial value increment value maximum value databank location.
BY ACCESS	List the following information on the specified sequence: idents with usage privilege.
BY REFERENCES	List the following information on the specified sequence: referencing objects.
FULL	List the following information about the specified sequence: initial value increment value maximum value databank location referencing objects idents with usage privilege comment creation date.

DESCRIBE SCHEMA Options

DESCRIBE SCHEMA	**Result**
BRIEF	List the following information about the specified schema: schema owner contained objects comment creation date.

DESCRIBE SHADOW

DESCRIBE SHADOW	**Result**
BRIEF	List the following information on the specified shadow: shadow creator databank name filename comment creation date

DESCRIBE COLLATION

DESCRIBE COLLATION	**Result**
BY REFERENCES	List the following information about the specified collation: columns using the specified collation all objects using the specified collation
FULL	List the following information on the specified collation: character set schema character set name pad attribute version delta definition idents with usage privilege on the specified collation comment creation date columns using the specified collation all objects using the specified collation

DESCRIBE SPECIFIC

Describe specific can be used to describe overloaded routines by using their specific name.

DESCRIBE SPECIFIC	**Result**
BRIEF	List the following information about the specified routine: schema name routine name statement type return type (if function) determinism access mode For each parameter: parameter name parameter mode data type
BY ACCESS	List the following information about the specified routine: idents with execute privilege on the routine
BY REFERENCE	Lists the following information about the specified routine objects referenced by the routine objects referencing the routine
FULL	Lists the following information about the specified statement schema name routine name statement type return type (if function) determinism access mode For each parameter: parameter name parameter mode data type idents with execute privilege on the routine objects referenced by the routine objects referencing the routine creation date comment

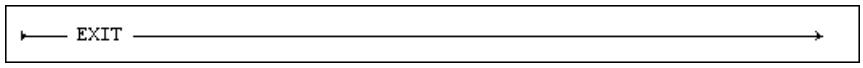
DESCRIBE STATEMENT

DESCRIBE STATEMENT	Result
BRIEF	List the following information about the specified statement: schema name statement name statement type statement definition
BY ACCESS	List the following information about the specified statement: idents with execute privilege on the statement
BY REFERENCE	Lists the following information about the specified statement: objects referenced by the statement
FULL	Lists the following information about the specified statement: schema name statement name statement type statement definition idents with execute privilege on the statement objects referenced by the statement creation date comment

EXIT

Leave BSQL.

Syntax



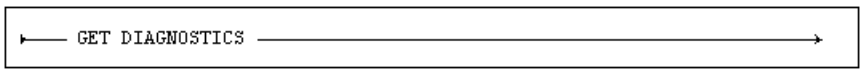
Description

Terminates the BSQL session.

GET DIAGNOSTICS

Get diagnostics for statement.

Syntax



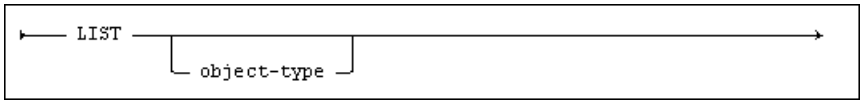
Description

Presents all status and diagnostics information for the preceding statement.

LIST

Lists information on a specified object.

Syntax



Where `object-type` is one of the object types listed below.

Description

The `LIST` command presents the following menu, if no object-type is specified:

```
                    Menu for List

 1. Databanks         8. Views             15. Shadows
 2. Domains           9. Modules           16. Collations
 3. Idents           10. Procedures        17. Methods
 4. Indexes          11. Functions         18. Specifications
 5. Objects          12. Triggers          19. Types
 6. Synonyms         13. Sequences         20. Statements
 7. Tables           14. Schemata           0. Exit
```

Choosing an item presents a submenu for choosing between different `LIST` functions - see the table that follows for details.

Entering an exclamation mark (`!`) in the Select field returns to the previous menu level. Entering a double exclamation mark (`!!`) returns two levels.

Giving an object type in the command executes the first menu choice for that type.

Selection numbers should not be used in script files, because the may change in future versions.

Note: `LIST` is not available when connected to a Mimer SQL server of version 8.1 or older.

LIST COLLATIONS Options

LIST COLLATIONS	**Result**
ALL	Lists all collations in the database.
IN SCHEMA	Lists collations in the specified schema.

LIST DATABANKS Options

LIST DATABANK	**Result**
ALL	Lists all databanks in the database.
CREATED BY	Lists databanks created by a specified ident.

LIST DOMAINS Options

LIST DOMAINS	**Result**
ALL	Lists all domains in the database.
IN SCHEMA	Lists domains in the specified schema.

LIST FUNCTIONS Options

LIST FUNCTIONS	**Result**
ALL	Lists all the functions the current ident has execute privilege on.
IN SCHEMA	Lists functions in the specified schema.

LIST IDENTS Options

LIST IDENTS	**Result**
ALL	Lists all idents in the database.
CREATED BY	Lists idents created by a specified ident.

LIST INDEXES Options

LIST INDEXES	**Result**
ALL	Lists the secondary indexes in the database.
IN SCHEMA	Lists secondary indexes in the specified schema.

LIST MODULES Options

LIST MODULES	**Result**
ALL	Lists all the modules in the database that are visible to (i.e. created by) the current ident.

LIST OBJECTS Options

LIST OBJECTS	**Result**
ALL	Lists objects in the database.
CREATED BY	Lists objects created by a specified ident.
WITH TYPE	Lists objects of a specified type.

LIST PROCEDURES Options

LIST PROCEDURES	**Result**
ALL	Lists all the procedures the current ident has execute privilege on.
IN SCHEMA	Lists procedures in the specified schema.

LIST SCHEMATA Options

LIST SCHEMATA	Result
ALL	Lists schemata created by the current ident.

LIST SEQUENCES Options

LIST SEQUENCES	Result
ALL	Lists all the sequences the current ident has usage privilege on.
IN SCHEMA	Lists sequences in the specified schema.

LIST SHADOWS Options

LIST SHADOWS	Result
ALL	List shadows created on databanks created by the current ident or all shadows if the current ident has shadow privilege.

LIST STATEMENTS Options

LIST STATEMENTS	Result
ALL	List all the precompiled statements the current ident has usage privilege on.
IN SCHEMA	List all statements belonging to the defined schema.

LIST SYNONYMS Options

LIST SYNONYMS	Result
ALL	Lists synonyms in the database.
IN SCHEMA	Lists synonyms in the specified schema.

LIST TABLES Options

LIST TABLES	Result
ALL	Lists tables in the database.
IN SCHEMA	Lists tables in the specified schema.

LIST TRIGGERS Options

LIST TRIGGERS	Result
ALL	List triggers defined on tables accessible to current user.
IN SCHEMA	Lists triggers in the specified schema.

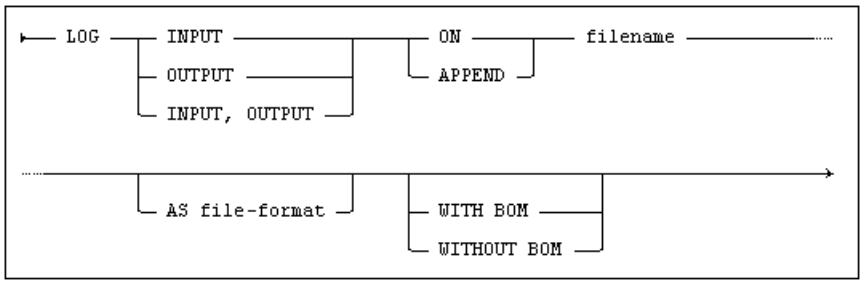
LIST VIEWS Options

LIST VIEWS	Result
ALL	Lists views in the database.
IN SCHEMA	Lists views in the specified schema.

LOG

Logs input, output or both to a specified sequential file.

Syntax



Description

All input, output or both will be logged in the specified sequential file.

If ON is specified a new file will always be created, otherwise the log data is appended to the file.

Logging is paused with the SET LOG OFF command and is resumed with the SET LOG ON command. Use CLOSE to stop logging permanently.

Using the AS option, you can set the file format to LATIN1, UTF8, UTF16, UTF16BE, UTF16LE, UTF32, UTF32BE or UTF32LE.

WITH BOM and WITHOUT BOM can be used to override platform specific default BOM behavior.

When logging INPUT the max width of logged data is determined by the value of LINEWIDTH.

When logging INPUT, OUTPUT the max width and max length of logged data is determined by the values of LINEWIDTH and LINECOUNT.

When logging OUTPUT the max width and max length of logged data is determined by the values of PAGEWIDTH and PAGELENGTH.

See the *SET LINECOUNT* on page *154*, *SET LINEWIDTH* on page *155*, *SET PAGELENGTH* on page *157* and SET*SET PAGEWIDTH* on page *157* for details.

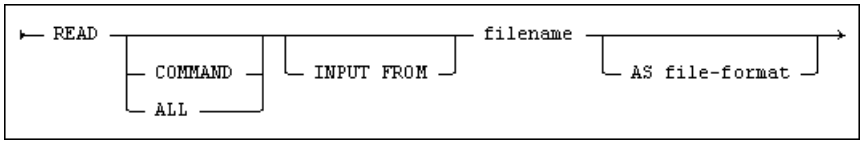
Example

```
SQL>log input,output on 'create_environment_log.dat';
```

READ INPUT

Reads commands from a sequential file.

Syntax



Description

Commands and SQL statements are read from the specified file.

When `READ ALL`, both commands and prompt answers are read from the sequential file. (`READ ALL` is default mode.)

When `READ COMMAND` is specified, commands are read from the input file while prompt answers are taken from the script file or interactively (depending on the situation).

Using the `AS` option, you can set the file format to `LATIN1`, `UTF8`, `UTF16`, `UTF16BE`, `UTF16LE`, `UTF32`, `UTF32BE` or `UTF32LE`.

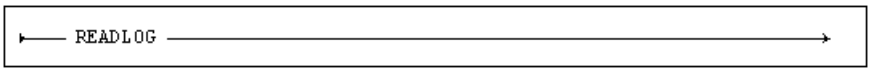
Example

```
SQL>read input from 'create_environment.dat';
```

READLOG

Obtains information about logged operations.

Syntax



Description

READLOG is a Mimer SQL function which enables you to read the contents of `LOGDB` so that you can check logged operations performed on the database since the last backup copy or incremental backup was taken.

You can use READLOG as an audit trail or, in the event of a system failure, to determine which databanks need to be restored (i.e. which databanks have been altered since the last backup).

Functions

READLOG enables you to select information from `LOGDB` on the basis of time interval, ident performing the operation, and specified databanks or tables.

This is particularly useful in production systems where `LOGDB` can contain a large number of entries.

Authorization

To list the log for selected tables or all tables in a databank, you must have `SELECT` access on the tables in question.

To list the log for the entire database, you must have `BACKUP` privilege.

Using the READLOG Functionality

You control the READLOG functionality using a menu from which different listing options may be set before finally performing the read operation.

The different listing options are set by using menu selections 1-5. The menu is re-displayed after selecting any of these so that further options may be set for the listing.

When all the desired listing options have been set in this way, a listing is produced from the log by choosing menu selection 6, 7 or 8 from under `List operations`.

```
                  -- Read log --

List definitions    List restrictions   List operations
----------------    -----------------   ---------------
1. Log file         3. Time interval    6. Specified tables
2. File properties  4. Ident            7. Tables in databank
                    5. Databank         8. All (no data)
0. EXIT
```

List Definitions – Output Control

Log File

Choosing `Log file` allows you to specify the name of a sequential file into which the listing is to be placed. In systems where the terminal may be addressed by a logical filename, this may be given to display the listing on the terminal.

If this option is not selected, a sequential file with the default name RDLOGL will be used.

The following example sets the log file explicitly:

```
SQL>READLOG;

                    -- Read log --

List definitions    List restrictions   List operations
----------------    -----------------   ---------------
1. Output           3. Time interval    6. Specified tables
2. Properties       4. Ident            7. Tables in databank
                    5. Databank         8. All (no data)
0. EXIT

Select: 1
Output to file or terminal (F/T) [F]: F
Log list file: READLOG.DAT
```

List Properties

The `file properties` choice is used to set either the width of a report or the format for the log file. The file format can be one of the following encodings:

```
1. Default
2. Latin 1
3. UTF 8
4. UTF 16
5. UTF 16 big endian
6. UTF 16 little endian
7. UTF 32
8. UTF 32 big endian
9. UTF 32 litte endian
```

Default means the encoding specified by the locale settings for the client.

List Restrictions

Time interval

This option allows the listing to be restricted to a given time interval, specified as a starting time and a finishing time.

Times are given as a single parameter representing year, month, day, hour, minute and second in the format `YYYYMMDDHHMMSS`.

If an incomplete time specification is given (truncated from the right), the remaining parameters are taken as low for the starting time and high for the finishing time. Thus giving 200211 as both the starting and finishing time, lists the log from the beginning to the end of November 2002.

A default time value is assumed if no time interval is specified, or may be chosen for starting or finishing time by specifying a 'blank' time.

If no start time is specified, the time at the beginning of the log is assumed. If no end time is specified, the time at the end of the log is assumed.

If neither a start time nor an end time is specified, the following message is displayed:

```
** No time restriction
```

A selected time interval applies for all subsequent list operations in the current session until the time interval is reset.

A time interval of two months has been selected in the following example:

```
Select: 3
Format   : YYYYMMDDHHMMSS
Starttime: 201211
Endtime  : 201212
```

Ident

Selecting an ident restricts the listing to operations performed by that ident. Only one ident may be selected for a given listing.

The default setting lists operations performed by all idents.

The default applies if no ident restriction is selected, or may be chosen by specifying a blank ident. If the default is chosen, the following message is displayed:

```
** No ident restriction
```

A selected ident applies for all subsequent list operations in the current session until a new ident is specified.

Example:

```
Select: 4
Identname: mimer_store
```

Databank

Selecting a databank restricts the listing to operations performed on that databank. This option must be specified if the list operation 7 (`Tables in databank`) is to be used. Only one databank may be selected for a given listing.

If no databank is specified, the list operation is done for all databanks. If this is the case, the following message is displayed:

```
** No databank restriction
```

A selected databank applies for all subsequent list operations in the current session until the databank is reset.

Example:

```
Select: 5
Databank: mimer_orders
```

List Operations

Specified Tables

This option activates listing of the log for selected tables in the database.

As many tables may be specified as are required, with the table name qualified, if necessary, by the name of the schema to which it belongs.

If no schema is specified, the schema with the same name as the current ident is assumed.

Databank restrictions selected with option 5 are ignored if specified tables are selected. However, any ident and time restrictions selected with options 3 and 4 are applied.

The ident running `READLOG` must have `SELECT` access on the requested tables, otherwise the following message is displayed for the table in question:

```
** No select access on table
```

If a non-existent table is requested, the following message is displayed:

```
** No such table
```

Errors of this type do not abort the listing if valid and invalid requests are mixed in the same operation.

The list operation is activated by giving a blank response to the prompt for a table name when all the required tables have been specified, as in the following example:

```
Select: 6
Table: HOTELADM.EMPLOYEE
Table: HOTELADM.STAFF
Table: HOTELADM.SALARY
Table:
```

Note: The list operation can be interrupted by entering an exclamation mark `!`.

Tables in Databank

Operations on all tables in the databank specified under option 5 are listed. If no databank has been selected, the following message is displayed and the user must select a new option:

```
**Databank not entered
```

Time or ident restrictions selected with options 3 or 4 are applied.

Data is listed only for those tables to which the ident running `READLOG` has `SELECT` access.

Tables to which access is denied are indicated by the following message in the log list file:

```
Table <schema-name.table-name> - No select access
```

All (No Data)

This option lists logged operations without details of data records (see below). The ident running `READLOG` must have `BACKUP` privilege.

If the privilege is not held by the current ident the following error message is displayed:

```
**   AUTHORIZATION FAILURE
```

Output Format

The output from `READLOG` is divided into transactions, showing the date and time, the ident performing the transaction (with entered program idents where appropriate) and the number of database records read during the transaction.

Note: The output does not contain statements for reconstructing the logged operations - it is simply a documentary record of the transactions performed on the database

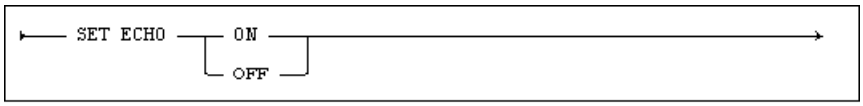
If list operations 6 or 7 (select by `Specified tables` or `Tables in databank`) are selected, the contents of the affected rows in the table are displayed. Insert and delete operations are listed as a single row. Update operations are recorded as the state of the row before and after the update.

If the list operation 8, `All (no data)`, is selected the operations are listed without the data records.

SET ECHO

Controls whether or not lines read during `READ INPUT` are echoed.

Syntax



Description

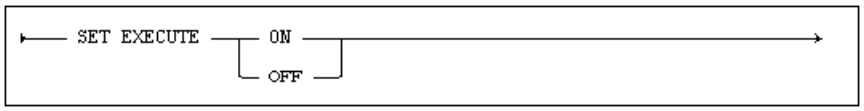
When echo is set to `ON`, lines read during `READ INPUT` are echoed to the BSQL window or log file. When echo is set to `OFF`, these lines are not echoed. The default value is `ON`.

The setting has no effect on the output of responses to BSQL commands and statements.

SET EXECUTE

Activate or deactivate the execution of queries.

Syntax



Description

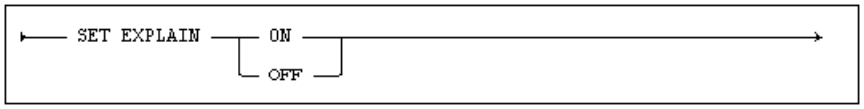
When execute is set to off no queries will be executed. This can be useful when using the explain facility or when testing a script for correctness.

Note: The `SET EXECUTE OFF` mode also affects statements like `CREATE INDEX` and `DROP INDEX`. I.e. do not forget to `SET EXECUTE ON` to be able to create or drop an index when examining different explain outputs for a query.

SET EXPLAIN

Activate or deactivate the explain facility.

Syntax



Description

When the explain facility is activated the execution plan for the query is shown. By default the query will be executed, to avoid this behavior the `SET EXECUTE` command can be used. (Note that `SET EXECUTE OFF` applies to DDL statements as well, e.g. `CREATE INDEX` will only verify correctness, no index will be created.)

The execution plan will show different operations and the sequence in which these operations are performed.

DbVisualizer Pro's explain plan tool is the recommended choice when working with a Mimer SQL server. See https://www.dbvis.com/features/tour/explain-plan/.

The BSQL Explain output for Mimer SQL is XML based. The different node types are described in the following table:

Node	Description
`cost`	This is the cost of executing the current node and all underlying nested nodes. The unit of cost is a ~ row access.
`hits`	Number of rows that are passed on from this part of the tree to higher nodes.
`visits`	Number of rows that the database server needs to process to find the requested rows.
`index`	This is the access path used to access the rows in the table.

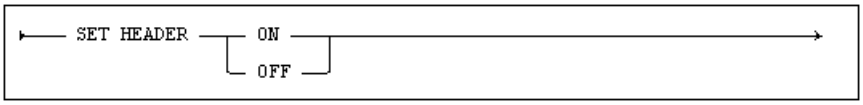
Node	Description
`type`	This is the type of the index. It can be primary key, secondary index or similar. Some constraints are maintained through the use of hidden indexes. In these cases the name corresponds to the constraint name.
`order`	Each table/index access is ordered according these numbers. The order can also be determined by viewing the explain tree.
`scan`	This shows whether the system can process the rows in the table efficiently or not. Sequential scan means the entire table/index must be scanned. Leadingkeys means there are conditions on one or several of the leading columns in the table/index. Trailingkeys means there are conditions on one or several trailing keys. In practise this usually means that all rows have to be processed unless the leading columns contain very few values. Unique means the row is uniquely identified and the can be processed quickly.
`index lookup only`	When index lookup only is used the base table does not need to be accessed as the index contains all the necessary information to process the query. If index lookup only is not used for each row in the index the corresponding row in the base table will be accessed.

How to read the XML output and how to understand the DbVisualizer explain is described in *Appendix A Mimer SQL Explain*.

SET HEADER

Activate or deactivate display of column headings.

Syntax



Description

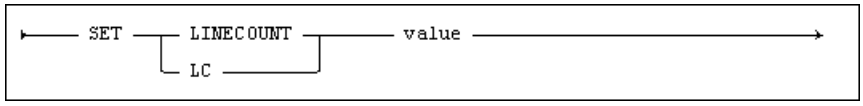
If `SET HEADER` is `ON`, column headings for result sets are displayed. If `SET HEADER` is `OFF` no column headings are displayed.

`SET HEADER ON` is default.

SET LINECOUNT

Sets the length of the BSQL window.

Syntax



Description

The `LINECOUNT` value defines the length of the BSQL window.

If `LINECOUNT` has a value greater than zero, output will temporarily be stopped after the number of lines defined for the value.

After the `Continue`-prompt, the user will have the choice of either continuing with the display or terminating the output.

Answering '`Y`' (default) implies that the output will continue until the number of lines is reached again.

Answering '`N`' terminates the output. Answering '`G`' will ignore the line count and the output will continue until all data are displayed.

If `LINECOUNT` is zero, the output will continue until all data is displayed.

The value of `LINECOUNT` must either be zero or >= 10.

The value of `LINECOUNT` affects data written to the terminal and data logged using the command `LOG INPUT,OUTPUT`.

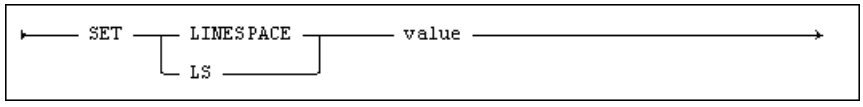
Default

If BSQL is run from a script job, `LINECOUNT` is zero by default. For interactive operation, the default value is environment-dependent.

SET LINESPACE

Sets the number of blank lines between each output record.

Syntax



Description

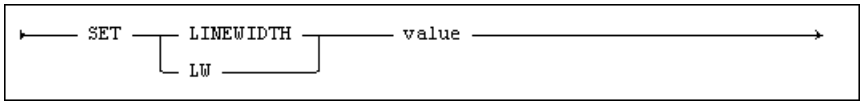
The `LINESPACE` value defines the number of blank lines to be written between each output record. This value is only used when printing the result of a `SELECT` statement.

The maximum value for `LINESPACE` is 9. The default value is 0.

SET LINEWIDTH

Specifies the width of the output.

Syntax



Description

The `LINEWIDTH` value defines the width for output to the BSQL window or log file. If a single line exceeds this value, the data will be continued at the next line.

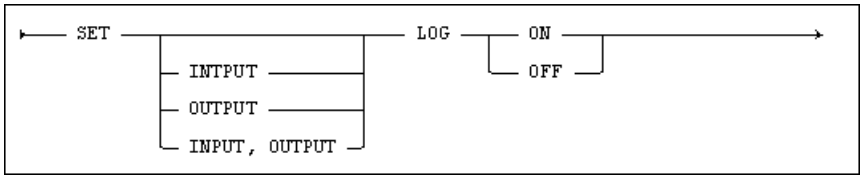
The value of `LINEWIDTH` affects data written to the terminal, and data logged using the commands `LOG INPUT` or `LOG INPUT,OUTPUT`.

The value for `LINEWIDTH` must be between 20 and 255.

SET LOG

Stops or resumes logging input, output or both.

Syntax



Description

When `SET LOG` is set to `OFF`, logging of input, output or both in a sequential file is temporarily stopped.

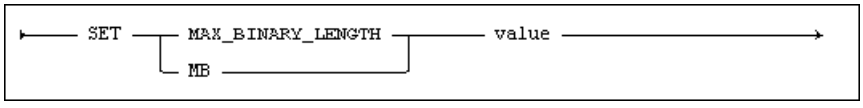
Resume logging with the `SET LOG ON` command.

If no input/output log is specified, all active logs are stopped or resumed.

SET MAX_BINARY_LENGTH

Specifies the maximum display length for binary columns.

Syntax



Description

The `MAX_BINARY_LENGTH` value defines the number of elements that are displayed when selecting data that is defined as binary, binary varying or binary large object.

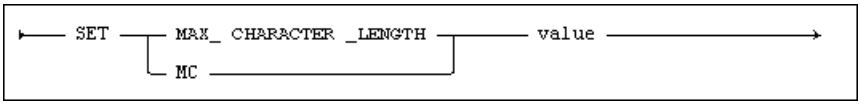
The default value is 15 000 and the value must be between 1 and 15 000.

As a binary string is shown as a hexadecimal string with two characters for each element the display length will be twice the value of `MAX_BINARY_LENGTH`.

SET MAX_CHARACTER_LENGTH

Specifies the maximum display length for character columns.

Syntax



Description:

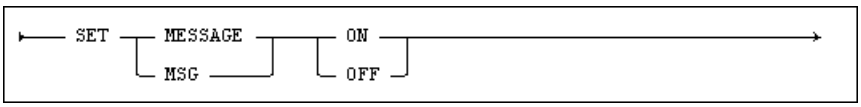
The `MAX_CHARACTER_LENGTH` value defines the number of characters that are displayed when selecting data that is defined as any character data type.

The default value is 15 000 and the value must be between 1 and 15 000.

SET MESSAGE

Specifies whether or not messages should be displayed.

Syntax



Description

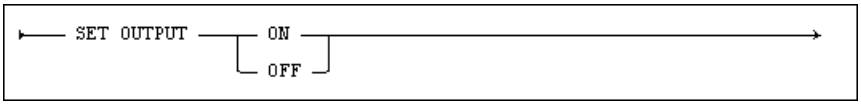
Specifies whether or not result messages such as `One row found` etc. are written.

The default setting is `ON`.

SET OUTPUT

Specifies whether or not output should be displayed.

Syntax



Description

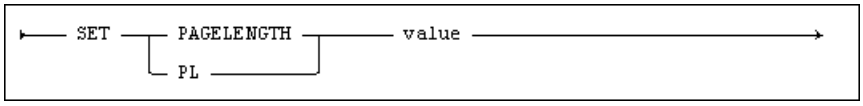
When `OUTPUT` is set to `ON`, the output from BSQL is written to the BSQL window. When it is set to `OFF`, the output does not appear.

The default value is `ON`.

SET PAGELENGTH

Specifies the page size of the output log file.

Syntax



Description

The `PAGELENGTH` value defines the page size of the file on which output is logged, i.e. at what interval a page break will be performed. A value of zero will result in no page breaks.

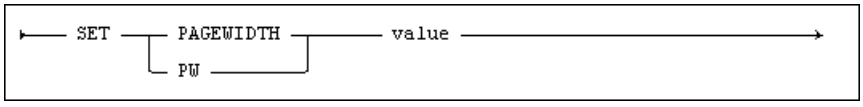
The `PAGELENGTH` value can either be set to zero or >= 10. The default value is machine-dependent.

The value of `PAGELENGTH` affects data logged using the command `LOG OUTPUT`.

SET PAGEWIDTH

Specifies the page width of the output log file.

Syntax



Description

The `PAGEWIDTH` value defines the width for output to the BSQL window or log file. If a single line exceeds this value, the data will be continued at the next line.

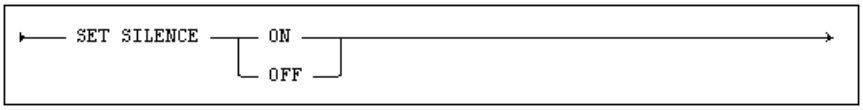
The value of `PAGEWIDTH` affects data logged using the command `LOG OUTPUT`.

The value for `PAGEWIDTH` must be larger than 20.

SET SILENCE

Specifies whether or not column headers or information messages should be displayed.

Syntax



Description

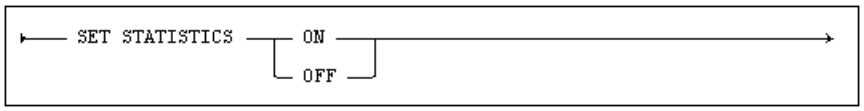
If `SET SILENCE ON` then BSQL will not display any column headers nor any messages when selecting or modifying data.

`SET SILENCE OFF` will revert to the default behavior.

SET STATISTICS

Specifies whether or not statement statistics should be displayed.

Syntax



Description

When `STATISTICS` is set to `ON`, additional information about `INSERT`, `DELETE`, `UPDATE` and `SELECT` statements will be shown. This includes number of table operations and transaction records.

A table operation is either a read or write to a table, index or temporary table that occurred during the statement. For instance, if a select statement read an index record and then reads additional columns (not present in the index) from the base table this will be counted as two table operations. Likewise when doing update statements, operations on base tables and operations on indexes will be counted as separate operations.

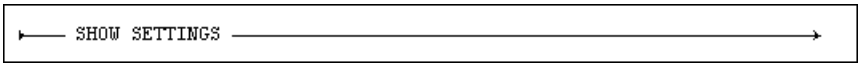
The number of transaction records for a statement includes the number of records insert, deleted or updated and any records that needs to be kept for checking constraints and transaction consistency.

The default value is `OFF`.

SHOW SETTINGS

Displays the current values of all set options.

Syntax



Description

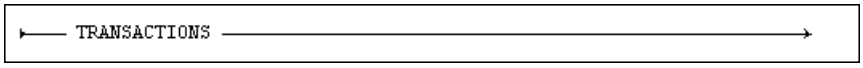
Displays the current values for all `SET` options, i.e. `ECHO`, `EXECUTE`, `EXPLAIN`, `LINECOUNT`, `LINESPACE`, `LINEWIDTH`, `LOG`, `MAX_BINARY_LENGTH`, `MAX_CHARACTER_LENGTH`, `MESSAGE`, `OUTPUT`, `PAGELENGTH`, `PAGEWIDTH`, `SILENCE`, `TRANSACTION START`, `TRANSACTION ISOLATION LEVEL`, `TRANSACTION MODE` (read only or read write).

Current server name, server version, and connection names are also displayed.

TRANSACTIONS

Displays the menu for administration of distributed transactions.

Syntax



Description

You can use the `TRANSACTIONS` command to monitor distributed transactions that are in a prepared or heuristically completed state. Note that all transactions are uniquely identified by the XID string. Because those strings are somewhat long, BSQL assigns a small sequence number to each line to be used as a shorthand. This shorthand is only valid until the `List transactions` option is used again. Note that since transactions are normally short-lived, the same transaction may be assigned different sequence numbers each time the `List transactions` option is used.

Note: The `TRANSACTIONS` command should only be used in exceptional circumstances, such as when a transaction monitor has crashed or a network failure has occurred making it is impossible to establish contact with a transaction monitor.

The command will present the following menu:

```
                    Menu for handling distributed transactions

     1. List transactions   2. Heuristic commit    3. Heuristic rollback
     0. Exit
```

The `List transactions` option displays a list of all distributed transactions that are either in a prepared ore heuristically completed state. For example:

```
NUMBER STATUS     XID
====== ========== ================================================================
     1 Prepared   34C6F6E675849446E616D6520
===
     2 Prepared   C6F6E675849446E616D6520
===

        2 transactions found
```

To heuristically commit or rollback a distributed transaction, you can choose option 2 or 3 in the menu. This will prompt for a transaction number which should correspond to a number in the listing. It is not possible to heuristically commit or rollback a distributed transaction without a prior listing.

When a distributed transaction is heuristically committed or rolled back it will remain in the list until it has been forgotten by the transaction monitor.

If the transaction with sequence number 1 (XID `34C6F6E675849446E616D6520`) was heuristically committed a subsequent listing would look like this:

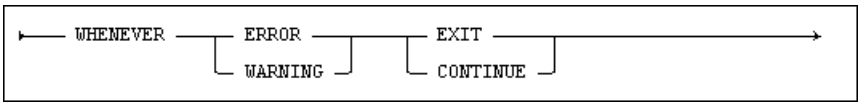
```
NUMBER STATUS     XID
====== ========== ================================================================
     1 Prepared   C6F6E675849446E616D6520
===
     2 Committed  34C6F6E675849446E616D6520
===

        2 transactions found
```

More information on distributed transactions in general, can be found in the *Mimer SQL Programmer's Manual, Chapter 10, Distributed Transactions*.

WHENEVER

Determines which actions should be taken in the event of an error or warning.

Syntax



Description

If an error or warning should occur in a file being run in a script, there are different action options that may be chosen to determine what should happen:

Execution Flow

`EXIT`	Leaves BSQL if script mode. Returns to prompt if interactive mode. I.e. if interactive mode and file input mode, the remaining file input is ignored and a new prompt is received.
`CONTINUE`	Continues execution.

Variables in BSQL

Host variables are used in SQL statements to pass values between the database and an application program, see the *Mimer SQL Programmer's Manual*.

Host variable syntax is also supported in BSQL to facilitate interactive design and testing of SQL statements intended for use in SQL application programs.

In BSQL, host variables serve as parameter markers, and the user is prompted for parameter values when the statement is executed.

You can use host variables used to:

- assign values to columns in the database (`UPDATE` and `INSERT` statements)
- to manipulate information taken from the database or contained in other variables (in expressions)
- to provide values for comparison predicates.

In all these contexts, the data type and length of the host variable must be compatible with that of any database values within the same syntax unit.

Writing Host Variables in SQL

Host variables are written in SQL as:

```
:host-identifier
```

or

```
:host-identifier :indicator-identifier
```

or

```
:host-identifier INDICATOR :indicator-identifier
```

In the first construction, the host identifier is the name of the main host variable.

In the second and third constructions, the main variable host-identifier is associated with an indicator variable indicator-identifier, used to signal the assignment of a null value to the main variable.

See the *Mimer SQL Programmer's Manual* for a description of the use of indicator variables.

Scope of Host Variables

The scope of host variables in BSQL is restricted to the individual usage instance in each statement.

Variables may not be used to pass values between separate statements, and the same variable name used more than once in a statement represents separate, independent variables.

Using Host Variables

When host variables are used in BSQL, BSQL prompts for the variable values, for example:

```
SQL>SELECT * FROM countries WHERE country = :COUNTRY;
COUNTRY: Spain
```

This corresponds to the statement:

```
SQL>SELECT * FROM countries WHERE country = 'Spain';
```

Note: The entered variable is not enclosed between single quotation marks, in contrast to the corresponding string value.
Variables enclosed in single quotation marks will be interpreted as literal strings.

Including Indicator Variables

If an indicator variable is included, you will be prompted for whether to use a null value.

If you answer the prompt with `No`, you will then be prompted for a value.

If you answer Yes, the null value will be used. For example:

```
SQL>UPDATE currencies SET exchange_rate = :RATE:IND
SQL&  WHERE code = 'BND';
Null ?n
RATE: 1.34

SQL>UPDATE currencies SET exchange_rate = :RATE:IND
SQL&  WHERE code = 'BND';
Null ?y
```

In the first example above, the exchange_rate value is updated to 1.34. In the second example, the exchange_rate value is set to null.

Note: The prompts appear in the order in which the variables are used in the statement.

BSQL and Multiple Connections

After logging in using BSQL, additional connections can be established using the CONNECT statement, which has the following form:

```
CONNECT TO 'database' [AS 'connection_name']
  USER 'username' USING 'password';
```

This statement establishes a new connection between the user and a database, see the *Mimer SQL Reference Manual, Chapter 12, CONNECT*for details.

The database may be given an explicit connection name for use in DISCONNECT and SET CONNECTION statements. If no explicit connection name is specified, the database name is used as the connection name.

Changing Connections

BSQL may make multiple connections to the same or different databases using the same or different idents, provided that each connection is identified by a unique connection name. In this situation only one connection is active and the other connections are inactive. A connection established by a successful CONNECT statement is automatically active.

A connection may be made active by the SET CONNECTION statement.

For example:

```
SET CONNECTION 'connection_name';
```

Disconnecting

The DISCONNECT statement breaks the connection between the user and a database. The connection to be broken is specified as the connection name or as one of the keywords ALL, CURRENT or DEFAULT.

```
DISCONNECT 'connection_name';
```

A connection does not have to be active in order to be disconnected. If an inactive connection is broken, BSQL still has uninterrupted access to the database through the current (active) connection, but the broken connection is no longer available for activation with SET CONNECTION.

If the active connection is broken, BSQL cannot access any database until a new CONNECT or SET CONNECTION statement is issued.

Note: The distinction between breaking a connection with DISCONNECT and making a connection inactive by issuing a CONNECT or SET CONNECTION for a different connection is, a broken connection has no saved resources and cannot be reactivated by SET CONNECTION.

The table below summarizes the effect on the connection con1 of CONNECT, DISCONNECT and SET CONNECTION statements depending on the state of the connection.

Statement	con1 non–existent	con1 current	con1 inactive
CONNECT TO 'DB1' AS 'CON1'	con1 current	error – connection already exists	error – connection already exists
DISCONNECT 'CON1'	error – connection doesn't exist	con1 disconnected	con1 disconnected
SET CONNECTION 'CON1'	error – connection doesn't exist	ignored	con1 made current
CONNECT TO 'DB2' AS 'CON2'	–	con1 made inactive	con1 unaffected
DISCONNECT 'CON2'	–	con1 unaffected	con1 unaffected
SET CONNECTION 'CON2'	–	con1 made inactive	con1 unaffected

Transaction Handling in Mimer BSQL

Normal Mimer SQL transaction handling behavior applies in Mimer BSQL. The default transaction start setting of implicit means that, by default, a transaction is started whenever one is needed.

For a detailed description of transaction handling behavior in Mimer SQL, refer to the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security*.

A special feature of BSQL is that all implicitly started transactions are automatically committed at the end of each statement, so that by default no attention needs to be paid to transaction handling at all in BSQL.

The START and COMMIT (or ROLLBACK) statements may be used together to group a number of statements into a single transaction when this is required.

Any transactions explicitly started using START will not be automatically committed by BSQL, so COMMIT or ROLLBACK must be used.

LOBs in BSQL

Although BSQL is not designed to handle Large OBjects (LOBs), it does provide limited LOB support for testing purposes.

A LOB is a column defined as being of type `Character Large OBject` (`CLOB`), `National Character Large OBject` (`NCLOB`) or `Binary Large OBject` (`BLOB`).

Columns defined as `CLOB` will, in all essentials, be treated as being the same as `VARCHAR` with a maximum length of 15 000 characters.

Columns defined as `NCLOB` will, in all essentials, be treated as being the same as `NCHAR VARYING` with a maximum length of 5 000 characters.

Columns defined as `BLOB` will be treated as `BINARY VARYING` with a maximum length of 15 000.

LOBs larger than these limits cannot be entered as input, and will be truncated as output.

For more information on `LOB` data types, see the *Mimer SQL Reference Manual, Chapter 6, Data Types in SQL Statements*.

Errors in BSQL

Error messages are shown when you attempt to execute an erroneous SQL statement. There are two types of errors: semantic errors and syntax errors.

Semantic Errors

Semantic errors arise when SQL statements are formulated with correct syntax, but do not reflect the user's intentions.

For example, suppose that a user wishes to select the string constant `Hotel:` and the actual hotel name from the table `HOTEL`, but uses double quotation marks instead of single quotation marks around the string constant:

```
SELECT  "Hotel:",NAME
FROM    HOTEL;
```

Double quotation marks are used to delimit identifiers containing special characters, so that the statement is interpreted as a request to select two columns, called `Hotel:` and `NAME`, from the table. The first column does not exist.

This example will in fact lead to an execution error, and is easily detected. Other semantic mistakes can be more difficult to find, when the statement is executed but gives the 'wrong' answer.

An example is the incorrect use of null in a search condition:

```
SELECT  RESERVATION FROM BOOK_GUEST
WHERE   CHECKOUT = CAST(NULL as DATE);
```

This will always give an empty result set, since null is not equal to anything.

The correct formulation would read `WHERE CHECKOUT IS NULL`.

Always check that the result of an SQL query looks reasonable, in particular if the query is complicated.

Syntax Errors

Syntax errors are constructions which break the rules for formulating SQL statements.

For example:

- spelling errors in keywords

  `SLEECT` (for `SELECT`)
- incorrect or missing delimiters

  `DELETEFROM` (for `DELETE FROM`)

  `SELECT column1 column2` (for `SELECT column1,column2`)
- incorrect clause ordering

  `UPDATE table WHERE condition SET values`

  (for `UPDATE table SET values WHERE condition`)

Syntactically incorrect statements are not accepted and an appropriate error message is displayed.

The error must be corrected before the statement can be executed.

For syntax errors, BSQL analyzes the statement and makes an intelligent guess as to where the error lies. This guess is based upon the most likely syntax or appearance of the statement in question. The system then points out the error and lists an error message based on this analysis. The appearance of this pointer on your screen is machine dependent. In the examples shown in this chapter, the pointer appears as '^'. The messages are self-explanatory.

The statement analysis is however not completely foolproof and misleading error messages may arise. If the message seems to be inaccurate, check the statement construction against the syntax diagram in the *Mimer SQL Reference Manual*.

Error Examples

Some examples of errors and resulting error messages are listed below.

Incorrect statement:

```
SELECT  AVG(country) FROM countries;
```

Error message:

```
SELECT  AVG(country) FROM countries;
           ^
Invalid operand type, expected type is NUMERIC or INTERVAL
```

Incorrect statement:

```
SELECT  country FROM countries
WHERE   currency_code ON ('USD','GBP','SEK');
```

Error message:

```
SELECT  country FROM countries
WHERE   currency_code ON ('USD','GBP','SEK');
                      ^
Syntax error, 'ON' assumed to mean 'IN'
```

Incorrect statement:

In the following example, the error analysis is misleading:

```
SELECT  country FROM countries
WJERE   currency_code = 'USD';
```

Error message:

```
SELECT  country FROM countries
WJERE   currency_code = 'USD';
        ^
Syntax error, END-OF-QUERY assumed missing
```

The misspelled word `WJERE` is not recognized as an attempt to write `WHERE`, so that the second line is not interpreted as a selection condition.

Error Messages

Error messages from BSQL are shown when you enter an illegal BSQL command or attempt to execute an erroneous SQL statement.

The error messages for erroneous SQL statements are the same as the return codes found in the *Mimer SQL Programmer's Manual*.

Error messages that can be received for illegal BSQL commands are:

Code	Message
-1	String exceeds 256 characters which is not allowed
-2	File could not be opened
-3	Too many files have been opened
-5	Conflict. One of COMMIT or ROLLBACK and EXIT or CONTINUE
-100	Undefined command <%>
-101	Ambiguous command <%>
-102	<%> command not valid in this context
-103	Missing semicolon
-104	Missing statement terminator (@)
-201	Syntax error
-202	Undefined keyword
-203	String expected
-204	Filenames must be enclosed in apostrophes
-205	Invalid numerical literal
-206	Unexpected end of command
-207	Too many parameters
-300	Failed to read dictionary

Code	Message
-400	Record too large for one page (<%> lines required) Increase value of LC/PL or set them to zero
-600	The number of host variables cannot exceed 20
-666	Space area exhausted
-700	Help databank not installed or inaccessible
-701	Help topic not found
-776	Maximum record length <%> exceeded
-777	Maximum header length exceeded
-800	Load/unload is not allowed within a transaction
-801	Pending transaction , Commit or Rollback
-802	Invalid transaction number, must be between 1 and <%>
-803	Server version and BSQL version must be the same when using READLOG
-804	The READLOG statement cannot be used within a transaction
-805	Invalid string literal, missing delimiter (')
-806	Error when reading from terminal
-807	<%> logging has not been activated with the LOG command
-810	There are no recovered XA transactions
-811	Support for XA commands is not enabled
-900	No buffer saved
-999	Too long statement
-1001	Syntax error in file name
-1002	File not found
-1003	File protection violation
-1004	File locked
-1005	Maximum number of opened files exceeded
-1006	Disk space exhausted
-1007	** Installation dependent **
-1008	** Installation dependent **
-1009	Unspecified file open error
-1101	Disk space exhausted
-1200	Previous perform file is not finished
-1300	Only select statements can be used with PRINT

Code	Message
-1400	Invalid numerical argument
-1500	Illegal value for <%>
-1600	The routine is overloaded. Use describe specific instead.
-1700	Maximum number of result items exceeded, limited to 300.
-1800	Maximum number of input items exceeded, limited to 300.

Appendix A

Mimer SQL Explain

The Mimer SQL server contains a highly advanced SQL optimizer. The optimizer performs numerous transformations and computes the most efficient access path to get the query results.

It is possible to view the results of the optimization process to help in the construction of efficient queries. In the bsql command line utility (aka. Batch SQL), the optimizer output may be viewed in its raw format. The output is xml-based.

It is also possible to view a graphical output of the same data in the DbVisualizer Pro front-end. (Note that this functionality is not available in the version bundled with the Mimer SQL distribution.) Download a DbVisualizer Pro trial license and try it out! (https://www.dbvis.com/download/)

Here is some reference material about explain output followed by some sample queries.

Node name	Explanation
`select`	The SQL statement is a select statement
`insert`	The SQL statement is an insert statement
`update`	The SQL statement is an update statement
`delete`	The SQL statement is a delete statement
`tempTable`	The optimizer has decided to use a temporary table for the results thus far in the query.
`union`	The optimizer is performing a union all of two or more result sets (i.e. concatenating the results without temporary table). An SQL UNION operation is typically translated to a tempTable containing a union.
`subselect`	A subselect node is a subquery such as an exists, a scalar subquery, or a quantified predicate (IN-clause etc.). A subquery will be executed for each row found in the outer table(s).
`constSubselect`	A constSubselect node is a subquery that has no dependencies to outer tables and returns zero or one row. Examples of this are non-correlated exists and scalar subqueries. These are only executed once and the result is reused for each row found in the outer table(s).
`innerJoin`	The innerJoin node is used when two tables are joined. Only rows that match the join condition are returned.

Node name	Explanation
outerJoin	The outerJoin node is used for an outer join query. When there is no match in the outer join, null values are returned.
crossJoin	This is a join where there are no join conditions between two tables. Every row in the first table will match all rows returned by the second table. The query may be lacking appropriate conditions when this occurs. Double check that your query returns the desired results.
table	This node is used to access the contents of a single table and/or index.

The following attributes are used:

Attribute	Used by	Explanation
name	table	Table name including alias used in SQL statement.
order	table	Scan order of the table/index in the query.
index	table	Here we see the name of the access path picked by the optimizer. The name depends on the type, either name of the index or the constraint. When a base table is read the primary key or internal key is always used.
scan	table	- sequential: this is a complete table scan. In general, they should be avoided as with more data in the table the query will take longer to execute. - trailingKeys: Means there are conditions on the second or later columns in the index. Unless all previous columns contain very few data values, this will be an expensive scan. - leadingKeys: One or several of the first columns in the index have conditions on them. Usually a good access path. - unique: The entire key is specified. Always a very good access path.
type	table	- Primary key. - Index created with a create index statement. - Foreign key index. - Unique constraint index. - Internal key. This is a generated key for tables without a primary key defined.

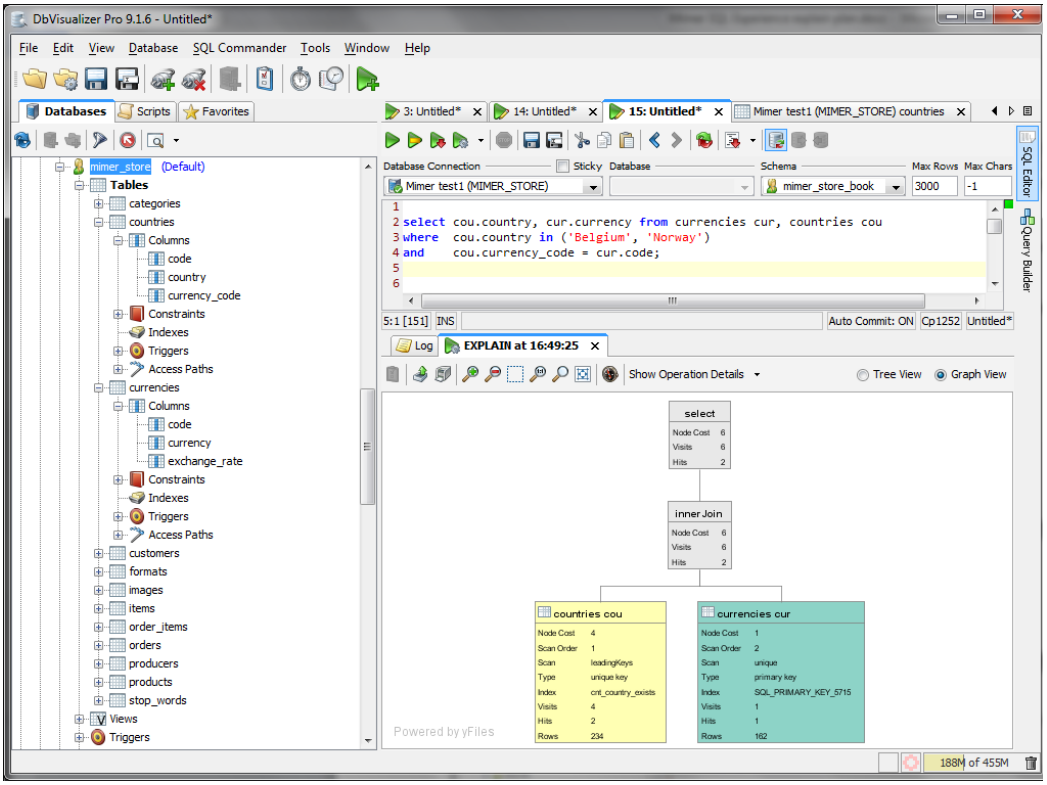
Attribute	Used by	Explanation
indexLookupOnly	table	When accessing an index, first the index is read and then the base table. However, if an SQL statement only uses column values that are present in the index then indexLookupOnly can be used. In this case only the index is read and no corresponding base table lookup is made. Please note that when a primary key or internal key is used, only the base table is read even though this option is not set.
cost	All nodes	The estimated cost for the accessing the table.
hits	All nodes	The number of hits that will be returned.
visits	All nodes	The number or rows that the system will read from a table or temporary table. When an index is accessed this includes both index table lookup and base table lookup (unless indexLookupOnly is used in which case the base table is never accessed). For parent nodes (such as a join node) this is the accumulated value for underlying accesses.
tempWrites	tempTable	Number of rows written to temporary table.
rows	table	This is the number of rows in the table. This is the value from the last time update statistics was run on the table. If there are no statistics collected for table the optimizer will look in the table to estimate the number of rows.

Join

Let us start with a simple query. It uses the Mimer sample database (see *The EXLOAD program* on page *180*.) Login as MIMER_STORE with password GoodiesRUs if you want to try it:

```
select cou.country, cur.currency from currencies cur, countries cou
where  cou.country in ('Belgium', 'Norway')
and    cou.currency_code = cur.code;
```

The query returns which currency is used in Belgium and Norway. The explain plan in DbVisualizer looks as follows:



The diagram is read starting in the lower left corner and then working your way up to the right. If in doubt, check the scan order field to see the execution order. The same query in bsql is shown here. Note the command `set explain on` is given first to see the explain plan:

```
SQL> set explain on;
SQL> select cou.country, cur.currency from currencies cur, countries cou
SQL& where  cou.country in ('Belgium', 'Norway')*
SQL& and    cou.currency_code = cur.code;
Start of explain result

  <select cost="6" hits="2" visits="6">
    <innerJoin cost="6" hits="2" visits="6">
      <table name="countries cou" order="1" index="cnt_country_exists"
             scan="leadingKeys" type="unique key"
             cost="4" hits="2" visits="4" rows="234"/>
      <table name="currencies cur" order="2" index="SQL_PRIMARY_KEY_5715"
             scan="unique" type="primary key"
             cost="1" hits="1" visits="1" rows="162"/>
    </innerJoin>
  </select>

End of explain result

country
currency
================================================
Belgium
Euros
===
Norway
Norwegian Kroner
===

                    2 rows found
```

Let us examine the explain output in some detail. Note that the XML contains exactly the same information as is used by DbVisualizer to display the explain graph.

The join order picked by the optimizer is to start with the countries table. This table is then joined with the currencies table.

```
<table name="countries cou" order="1" index="cnt_country_exists"
       scan="leadingKeys" type="unique key" cost="4" hits="2" visits="4"
rows="234"/>
```

Order 1 shows that the countries table is read first. The unique key cnt_country_exists index is used to scan the table. We have a condition on the first column in the index (cou.country = 'Belgium') which is why the scan is "leadingKeys". In DbVisualizer you can see all the columns of cnt_country_exists under Access Paths:

The index cnt_country_exists has both the country column and the primary key code column. The visit count is 4 because two rows are read in the index, and two rows from the base table. This will result in a hit-count of 2 rows. The statistics indicates there are 234 rows in the countries table.

Let us now examine the join-node:

```
<innerJoin cost="6" hits="2" visits="6">
  <table name="countries cou" order="1" index="cnt_country_exists"
scan="leadingKeys"
         type="unique key" cost="4" hits="2" visits="4" rows="234"/>
  <table name="currencies cur" order="2" index="SQL_PRIMARY_KEY_5715"
scan="unique"
         type="primary key" cost="1" hits="1" visits="1" rows="162"/>
</innerJoin>
```

The join node contains the cost of processing the two tables. The number of visits and hits are as follows:

```
Visits (6) = visits in countries (4) + hits in countries (2) * visits in currencies
(1)
Hits (2) = hits in countries (2) * hits in currencies (1)
```

When there are no temporary tables involved the cost is equal to the total number of visits.

Let us see what would happen if we were to force the opposite join order. This is done by using the `{order}` clause in the from-list:

```
SQL> set explain on;
SQL> select cou.country, cur.currency
SQL& from {order} currencies cur, countries cou
SQL& where  cou.country in ('Belgium', 'Norway')
SQL& and    cou.currency_code = cur.code;
Start of explain result

  <select cost="486" hits="162" visits="486">
    <innerJoin cost="486" hits="162" visits="486">
      <table name="currencies cur" order="1" index="SQL_PRIMARY_KEY_5715"
             scan="sequential" type="primary key"
             cost="162" hits="162" visits="162" rows="162"/>
      <table name="countries cou" order="2" index="SQL_FOREIGN_KEY_5723"
             scan="leadingKeys" type="foreign key index"
             cost="2" hits="1" visits="2" rows="234"/>
    </innerJoin>
  </select>

End of explain result
```

This was clearly a bad idea. We now got a sequential scan of the currencies table. Since we only had a join condition (`cou.currency_code = cur.code`) on this table which cannot be evaluated until we read both tables we get 162 hits. The countries table now uses the foreign key index to find the values. The inner join cost was:

```
Visits (486) = visits in currencies (162) + hits in currencies (162) * visits in
countries (2)
Hits (162) = hits in currencies (162) * hits in countries (1)
```

Inner join, outer join and cross join are all computed in this way.

Temporary Tables

There are several different types of temporary tables depending on which operation is being handled. For example distinct, group by and order by. Tables used for order by and group by are both inserted to and read. This is seen as both a write count and a visit count. Distinct temporary tables are only used to avoid duplicates and never have to be read. In this case we have a write count, but no visit count. It is a bit more expensive to perform write operations than read operations. The optimizer currently uses a factor of ~1.4 to estimate the cost.

Let us look at an example:

```
SQL> set explain on;
SQL> select cou.country from currencies cur, countries cou
SQL& where    cou.currency_code = cur.code
SQL& and cur.currency in ('Swiss Francs', 'Pulas', 'Czech Korony')
SQL& order by cou.country;
Start of explain result

  <select cost="175" hits="3" visits="171" tempWrites="3">
    <tempTable cost="172" class="TempTableOrderBy" hits="3" visits="3"
tempWrites="3">
      <innerJoin cost="168" hits="3" visits="168">
        <table name="currencies cur" order="1" index="SQL_PRIMARY_KEY_5715"
               scan="sequential" type="primary key"
               cost="162" hits="3" visits="162" rows="162"/>
        <table name="countries cou" order="2" index="SQL_FOREIGN_KEY_5723"
               scan="leadingKeys" type="foreign key index"
               cost="2" hits="1" visits="2" rows="234"/>
      </innerJoin>
    </tempTable>
  </select>

End of explain result
```

As can be seen the optimizer estimates that 3 rows will be both written and read from the temporary table. The cost (172) = innerjoin cost (168) + tempWrites (3) * 1.4. In the next level we add the cost for reading (visits="3") so the total cost is 175.

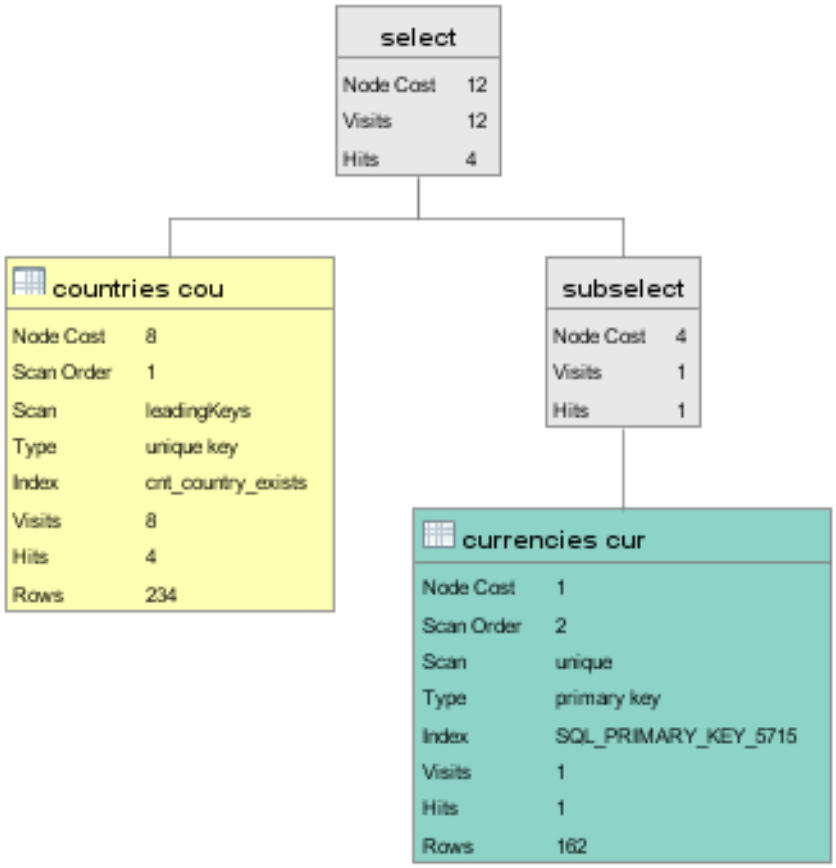
Subqueries

There are three variations of optimization of subqueries that are important to understand:

- The simplest one is where the subquery is executed once for each row of the outer query.
- Sometimes it is possible to execute the subquery as a join with the outer table(s). In this case the subquery can occur somewhere in the join order. Depending on the actual conditions a temporary table may be needed to eliminate duplicates in this case.
- If the subquery is not correlated, i.e. has no conditions that relate to the outer query and only return zero or one rows, then the query can be executed once and for all and the result is then reused as the outer tables are processed.

We will look at examples of the above three cases.

```
select cou.country from countries cou
where country in ('Andorra','Angola','Anguilla','Antigua and Barbuda')
and   cou.currency_code in (select cur.code from currencies cur
                            where exchange_rate > 0.3);
```



For each hit in the countries table we run the subselect. In the query the optimizer has used the primary key to look up the corresponding row in the currencies table. The subselect is evaluated 4 times and each has a cost of 1.

We will now take a look at a query where the subquery participates in the outer query join.

```
SQL> set explain on;
SQL> select cur.currency
SQL& from currencies cur
SQL& where exists (select 1 from countries c
SQL&               where c.country in ('Sweden' ,'Japan')
SQL&               and c.currency_code = cur.code);
Start of explain result

  <select cost="9" hits="2" visits="6" tempWrites="2">
    <tempTable cost="9" class="TempTableJoinBySubselect" tempWrites="2">
      <innerJoin cost="6" hits="2" visits="6">
        <table name="countries c" order="1" index="cnt_country_exists"
               scan="leadingKeys" type="unique key"
               cost="4" hits="2" visits="4" rows="234"/>
        <table name="currencies cur" order="2" index="SQL_PRIMARY_KEY_5715"
               scan="unique" type="primary key"
               cost="1" hits="1" visits="1" rows="162"/>
      </innerJoin>
    </tempTable>
  </select>

End of explain result
```

In the explain output the exists now participates as part of the join with the currencies table. An extra temporary table is in this query needed to eliminate duplicates.

And finally the third type is the constant subquery.

```
SQL> set explain on;
SQL> select cur.currency
SQL& from  currencies cur
SQL& where cur.exchange_rate < (select exchange_rate
SQL&                            from currencies xcur, countries cou
SQL&                            where  cou.country = 'Belgium'
SQL&                            and    cou.currency_code = xcur.code);
Start of explain result

  <select cost="165" hits="54" visits="165">
    <constSubselect cost="3" hits="1" visits="3">
      <innerJoin cost="3" hits="1" visits="3">
        <table name="countries cou" order="1" index="cnt_country_exists"
               scan="leadingKeys" type="unique key"
               cost="2" hits="1" visits="2" rows="234"/>
        <table name="currencies xcur" order="2" index="SQL_PRIMARY_KEY_5715"
               scan="unique" type="primary key"
               cost="1" hits="1" visits="1" rows="162"/>
      </innerJoin>
    </constSubselect>
    <table name="currencies cur" order="3" index="SQL_PRIMARY_KEY_5715"
           scan="sequential" type="primary key"
           cost="162" hits="54" visits="162" rows="162"/>
  </select>

End of explain result
```

The select using currencies and countries is a scalar subselect that only returns a single value. There are no references to the outer table currencies cur. Therefore the constSubelect node can be used. It is executed only once, and then the result is used for each row in the currencies cur table. In the query above the cost is much cheaper than evaluating the subquery for each row in currencies cur (instead of a cost of 165 we would get 660 = 165 + 165 * 3).

Union

Let us look at an example of how union costs are computed:

```
SQL> set explain on;
SQL> select c.category from categories c
SQL& union
SQL& select f.format from formats f;
Start of explain result

  <select cost="30" hits="13" visits="16" tempWrites="10">
    <tempTable cost="30" class="TempTableUnion" visits="3" tempWrites="10">
      <union cost="13" hits="13" visits="13">
        <table name="formats f" order="1" index="fmt_primary_key"
               scan="sequential" type="primary key"
               cost="10" hits="10" visits="10" rows="10"/>
        <table name="categories c" order="2" index="ctg_primary_key"
               scan="sequential" type="primary key"
               cost="3" hits="3" visits="3" rows="3"/>
      </union>
    </tempTable>
  </select>

End of explain result
```

In a union duplicates are eliminated. This can be seen as a tempTable node. The cost of the union is the cost of each branch and then the temporary table handling. Remember that writes cost an extra ~1.4 in the cost. It can be noted that the optimizer has reordered the union branches because the select from the categories tables only returns distinct values because there is a unique index on the category column. When the last union branch is distinct (i.e. that branch alone does not return any duplicate values), the system will only check if the value has been seen in earlier union branches. This can be seen as tempWrites="10" (from the first union branch) and visits="3" to check that the value has not occurred before. The total cost (30) = union cost (13) + tempWrites 10 * 1.4 + temp table visits (3).

The optimizer does not know anything about the actual content of the two columns. However, the SQL programmer may know that the values in the two tables are distinct. If that is the case the query can be rephrased as:

```
SQL> set explain on;
SQL> select c.category from categories c
SQL& union all
SQL& select f.format from formats f;
Start of explain result

  <select cost="13" hits="13" visits="13">
    <union cost="13" hits="13" visits="13">
      <table name="categories c" order="1" index="ctg_primary_key"
             scan="sequential" type="primary key"
             cost="3" hits="3" visits="3" rows="3"/>
      <table name="formats f" order="2" index="fmt_primary_key"
             scan="sequential" type="primary key"
             cost="10" hits="10" visits="10" rows="10"/>
    </union>
  </select>

End of explain result
```

The change is that `UNION ALL` is used instead of `UNION`. This will, of course, result in a more efficient query as no temporary table handling is needed. The cost has subsequently dropped from 30 to 13!

Appendix B

The Example Environment

This appendix describes the contents of the Mimer SQL example environment.

The complete environment is owned by a `USER` ident named `MIMER_STORE`.

The `MIMER_STORE` ident is granted the privilege to create databanks and idents.

The example environment is based on a store that sells music and books, available in various formats. The store also has a web site, through which users can place orders. Stored procedures (SQL/PSM) are used to take care of various aspects of the business logic.

The example environment forms as the basis for examples given in the Mimer SQL documentation set.

Linux: On Linux, the `dbinstall` utility, used to create the system databanks, provides an option to create the example environment.

Win: On Windows, the Mimer Administrator is used to create the system databanks. The wizard used for this provides an option to create the example environment.

The example environment can also be created manually by using the `EXLOAD` program (see below).

For more information about the example environment, see the feature description 'The Example Database' on the Mimer SQL developer site, https://developer.mimer.com/article/the-example-database/.

The EXLOAD program

By using the EXLOAD program the example environment can be created or removed. (The example environment can also be removed by dropping the MIMER_STORE ident from any SQL command tool).

At installation of the example environment, default values are provided for passwords and for databank filenames. These default values can be changed by using the ALTER command after the installation is completed or by explicitly giving passwords and filenames as command line arguments to the EXLOAD command, see the syntax below.

Syntax

The EXLOAD syntax (expressed in Unix-style) is as follows:

```
exload [-s | -m] [-f | -r] [-p pass] [database]

exload [--single | --multi] [--force | --remove] [--password=pass] [database]

exload [-v|--version | [-?|--help]
```

When creating the example database it is possible to change the default values for passwords and filenames. This is done by adding parameters after the 'database' parameter (that must be given in this case). These optional parameters must be added in the following order:

```
[MIMER_STORE-password
 MIMER_ADM-password
 MIMER_USR-password
 MIMER_WEB-password
 MIMER_STORE-databank-filename
 MIMER_ORDERS-databank-filename
 MIMER_BLOBS-databank-filename]
```

Command-line Arguments

Unix-style	VMS-style	Function
-s --single	/SINGLE	Single-user mode database access
-m --multi	/MULTI	Multi-user mode database access
-r --remove	/REMOVE	Remove the example database
-f --force	/FORCE	Install the example database even if it already exists. The old environment is dropped.
-p SYSADMpwd --password=SYSADMpwd	/PASSWORD=SYSADMpwd	Password for SYSADM
database	database	Database name
MIMER_STOREpwd	MIMER_STOREpwd	Password for MIMER_STORE ident
MIMER_ADMpwd	MIMER_ADMpwd	Password for MIMER_ADM ident
MIMER_USRpwd	MIMER_USRpwd	Password for MIMER_USR ident
MIMER_WEBpwd	MIMER_WEBpwd	Password for MIMER_WEB ident
MIMER_STOREdbfilename	MIMER_STOREdbfilename	Filename for MIMER_STORE databank
MIMER_ORDERSdbfilename	MIMER_ORDERSdbfilename	Filename for MIMER_ORDERS databank
MIMER_BLOBSdbfilename	MIMER_BLOBSdbfilename	Filename for MIMER_BLOBS databank
-? --help	/HELP	Show help text.
-v --version	/VERSION	Display version information.

All parameters to EXLOAD are optional. If no command line arguments are given, the following will happen:

- A SYSADM password is needed and will be prompted for.
- Access mode and database name will be taken from MIMER_MODE and MIMER_DATABASE, respectively.
- If the example environment is already installed a question is prompted whether to continue or not
- Default values are used for all example database passwords and filenames. The values used are displayed when the EXLOAD program is executed.

Exit Codes

The EXLOAD program returns an error status to the operating system when an error is encountered. This may be useful when running EXLOAD from scripts.

The following error codes are used:

Linux/Windows	OpenVMS	Usage
0 (success)	1 (success)	This code is used when EXLOAD has executed successfully.
>1 (error)	2 (error)	This error code is used when EXLOAD failed to execute successfully.

The MIMER_STORE Schema

In this base schema most of the objects supported by Mimer SQL are introduced: collations, databanks, domains, idents, sequences, synonyms, tables and views. They are used to demonstrate some of the functionality that is available – constraints; primary, unique and foreign keys; secondary indices; NOT NULL; default values; and column and table check clauses. The schema also includes examples of PSM routines and triggers.

When an ident, other than a GROUP, is created, a schema of the same name is created by default. Objects in the MIMER_STORE schema have to be created before other objects can reference them.

Databanks

A databank is the physical file where a collection of tables is stored. There are three databanks defined in the example environment; the databank where the tables containing BLOBs will be stored is defined with the TRANSACTION option. The databanks have names that start with the prefix MIMER_.

Domains

There are a number of domains defined. In essence, a domain is a column data type specification that can be used in a number of table column definitions.

A number of different data types are introduced. The domains introduce the idea of consistency in the table definitions and demonstrate how to use check clauses, including the equivalent of NOT NULL.

Sequences

A sequence is an object that can provide a unique integer value. The sequences start with initial values other than the default to give the various sequences different number ranges; this is sometimes helpful when looking at data. The sequence names have a _SEQ suffix to help to clarify when they are being used.

Tables

Data in a relational database is logically organized in tables, which consist of horizontal rows and vertical columns. Columns are identified by a column-name.

A relational database is built up of several inter-dependent tables that can be joined together. Tables are joined by using related values that appear in one or more columns in each of the tables.

All the foundation tables are created under the `MIMER_STORE` schema; these tables provide a reasonably complete set for an introduction to SQL.

The tables `CURRENCIES` and `COUNTRIES` are used in many of the basic examples in this manual and contain entries for almost all countries and currencies, providing a moderate amount of data for any exercise.

Table name	Description
`CURRENCIES`	Holds currency details. Introduces the use of domains (qualified with the schema name) and the decimal data type. Demonstrates that column constraints can include the primary key definition. Introduces the concept of `NOT NULL` in column definitions. Uses the `CHARACTER` data type to show the difference compared to `CHARACTER VARYING`. The table is created with the schema name explicitly included, after this example the default option is used. Note the table is not defined in a specific databank, the system determines which databank to use; all other tables are created in a named databank. Examples of creating comments on the table and columns follow this table creation; these are the only examples of comments.
`COUNTRIES`	Holds country details. Introduces the concept of naming constraints; as when naming objects, a consistent approach will give benefits. Includes a unique constraint. Introduces a foreign key definition linking the currency code to an entry in the `CURRENCIES` table. The column in the referenced table (`CURRENCIES`) is explicitly named; in future this is only done if the columns are not defined as the primary key.
`CATEGORIES`	Product categories, i.e. Music and Books. A category of 'Video' is included for course work. Shows the use of a collation in a column definition. A collation is a set of rules that determines how data is sorted and compared. Character data is sorted using collations that define the correct character sequence, with the capability to handle case-sensitivity and accents. In this table create statement the primary, unique and foreign keys are defined as table constraints. This is the style that has been adopted in the rest of the schema, the previous examples were to show the syntax possibilities; again a consistent approach will make it easier for others.

Table name	Description
FORMATS	Product formats, e.g. Paperback, Audio CD. A domain could/should have been used for the CATEGORY_ID columns in the CATEGORIES and FORMATS table. There are two unique constraints, both defined on two columns. A DISPLAY_ORDER column has been included; the idea is that this can be used to display the different formats for a particular product in an order other than alphabetic.
PRODUCERS	Name of the record label or book publisher i.e. the organization that made the product. Shows the use of a column default value that refers to a sequence.
PRODUCTS	Name of the product, i.e. name of the album or title of the book. It may be that the same product name is the title of a book and an album. Until this point these referenced tables had been defined with the primary key as the identifier and the associated 'name' as a unique key; this demonstrates the reverse. This table holds a Soundex value of the name. As we shall see later the name is processed before the Soundex value is taken. A secondary index is defined on this Soundex value.
IMAGES	An item may have an image associated with it; for example the image could be an album or book jacket cover. Note that this table is in the MIMER_BLOBS databank and therefore changes are not logged.

Table name	Description
ITEMS	This forms a link between a product name and the different formats in which it is available. For example an album may appear in a number of different formats: Audio CD, Cassette, DVD Audio and Vinyl. The table contains a number of attributes for the item, such as price and stock level. There is also a reorder level but that is an extension for the classroom (i.e. a trigger can be used to place an order). This table references a number of other tables and introduces the ability to specify what action to take when a referenced row is deleted. Both the PRODUCER_ID and IMAGE_ID permit a null value, which allows outer joins to be demonstrated. A date data type is introduced in RELEASE_DATE. A secondary index is created on the date column allowing examples of how date ranges can be used. The European Article Numbering (EAN) code is a useful example; it is the barcode that appears on everything these days. It is a unique key in its own right but the table has ITEM_ID as the primary key, making the EAN_CODE a candidate key. Obviously the EAN_CODE could have been used in referencing tables but in general it is a mistake to use external identifiers to join tables; this conflicts with the use of the International codes in the CURRENCIES and COUNTRIES tables but the data in these tables can be considered to be static. A property of an EAN code is that it is a 13-digit number that incorporates a check digit. A PSM function to validate the check digit is included later in this schema; an ALTER TABLE statement then uses the PSM routine in a check clause to validate the EAN code.
CUSTOMERS	Personal details for store customers, whether they are on a mailing list or are Web users. Shows the use of current_date as a default value against the REGISTERED column. The idea is that when customers order through the Web-site they will be prompted for their e-mail address and password; the table check clause makes sure that the e-mail and password are either both defined or neither is defined. Secondary indices are defined on the date of birth and post code columns, these are both useful when trying to identify a customer (e.g. when there are too many John Smiths to search through by name).
ORDERS	A record of when a customer placed an order.

Table name	Description
ORDER_ITEMS	What items went into an order and the price at the time that the order was placed. This is the first time that a multi-column primary key is used. VALUE is an SQL reserved word; the definition shows how to force the use of a reserved word.
STOP_WORDS	Contains the 100 most common words in the English language.

Note that the CUSTOMERS, ORDERS and ORDER_ITEMS tables are grouped together in the MIMER_ORDERS databank.

It could be argued that the ITEMS table is not fully normalized – the PRODUCER_ID is repeated for each different format. But if you think about it you will realize that there is no benefit in introducing another table, only a penalty.

PSM Routines

The MIMER_STORE schema also introduces some fairly simple PSM routines (defined in a module named ROUTINES). These routines provide a basic introduction to functions and procedures; they demonstrate the general syntax of a number of PSM statements, including deterministic and access options.

Routines defined in a module cannot be modified without dropping the module, in which case all the routines in the module are dropped.

Note: The use of the '@' character which is used in BSQL to delimit SQL statements whose syntax involves use of the normal end-of-statement character ';' before the actual end of the statement. The '@' character may be used to delimit any statement; this is useful when dealing with large statements as the error reporting facility in BSQL shows more information in such cases.

You can debug PSM routines using Mimer SQL's Java-based graphic debugger. The debugger has support for watching variables, step-wise execution and setting breakpoints. You can debug procedures, functions and triggers.

Routine name	Description
AGE_OF_ADULT	Procedure that returns the age that a person is considered to be an adult. A slightly contrived procedure to introduce the in and out parameter types; also includes an interval data type and a case statement.
CAPITALIZE	Function that processes a string making the first character of a word uppercase and the remainder of the word lowercase.
CAST_TO_DATE	Function that takes a character string (dd/mm/yy) and converts it to a date data type. Note the use of the like comparison. Introduce the concept of signaling, but this can be skipped over at this point.

Routine name	Description
EAN_CHECK_DIGIT	Function that returns the check digit for an EAN.
EXTRACT_DATETIME	Function to return just the variable component from a datetime or interval value. The value has to be cast to a character string in the call. Uses 'is not null'.
EXTRACT_DATE	Function to show how EXTRACT_DATETIME is used.
INDEX_TEXT	Function to process a character string, removing all punctuation characters. Also removes any of the words that are held in the table STOP_WORDS by using the exists predicate.
PRODUCT_SEARCH_CODE	Function to return the Soundex value of a character string after it has been processed by the INDEX_TEXT function.
RECIPIENT	Function to form the recipient details, e.g. for a letter address – Mr J Smith. Note that the use of 'is not null' is redundant.
SALUTATION	Function to create a formal or informal salutation based on the age of the person. Performs simple date arithmetic using the AGE_OF_ADULT function.
STOP_WORDS	Database load procedure to extract the words from a character string and insert them into the STOP_WORDS table.
VALIDATE_EAN_CODE	Function to return the EAN code with the correct check digit. Example of large number arithmetic.

The ITEMS table is altered to include a check clause using the VALIDATE_EAN_CODE function to ensure that only valid EAN codes can be entered.

After the definition of the views (see next section for details) there is another PSM routine, it shows how to create a standalone procedure.

Procedures

Procedure name	Description
COMING_SOON	Result set procedure that returns the products for a particular category that will be released in the next month. Makes use of the PRODUCT_DETAILS view to demonstrate that a select against a view is treated the same as a table. Shows how to define a result set procedure and one form of the row data type. Introduces the concepts of cursors. The use of get diagnostics avoids any real need to cover condition handlers at this stage.

Views

There are a number of views in the MIMER_STORE schema to show various aspects of the functionality that can be supported.

View name	Description
CUSTOMER_DETAILS	Based on the single table CUSTOMERS. Shows the simplest type of view – updatable if the user has the privilege. Allows a new customer to be added to the database.
SWEDISH_CUSTOMERS	Based on the previous view (i.e. views can be defined on views) but with a selection condition so that only Swedish customers are displayed. Uses a 'with check option' so that only Swedish customers can be operated on through this view.
CUSTOMER_ADDRESSES	Again based on the CUSTOMER_DETAILS view but also joined to the COUNTRIES table. Shows how PSM functions can be used from standard SQL statements, including applying pre-defined functions on the result. Demonstrates how to name the columns in a view definition. Introduces the coalesce expression. The join is slightly more complicated than it needs to be because the column names are not consistent.
PRODUCT_DETAILS	Forms a view across a number of tables, including an outer join on the PRODUCERS table. Note that the coalesce expression in the select list is named; all calculated columns in a view have to be named.

Triggers

There are examples of two very simple statement triggers in the MIMER_STORE schema, both defined against the PRODUCTS table. They are designed to ensure that the value of the PRODUCT_SEARCH column is based on the product name. The insert trigger unconditionally performs an update of the PRODUCT_SEARCH column (setting it to the default value), which will force the update trigger to fire.

The update trigger checks whether the value in the PRODUCT_SEARCH column is equal to the value returned by the PRODUCT_SEARCH_CODE function when applied to the product name. If the value of PRODUCT_SEARCH is inconsistent with the product name then an update is performed; this will cause a recursive call of the update trigger. Note that an update statement causes the trigger to fire even when no rows are updated, hence the use of 'if exists'.

The PRODUCT_SEARCH_CODE function returns the Soundex value of the product name after it has been processed by the INDEX_TEXT function, which basically involves stripping out any of the words that appear in the STOP_WORDS table (i.e. the 100 most common words in the English language). As an example, the album title 'Same As It Ever Was' is reduced to 'Same Ever' by the INDEX_TEXT function and the Soundex value returned by the PRODUCT_SEARCH_CODE function would then be '467900'. By using this function against a user input string, it is possible to allow a limited level of fuzzy matching.

Idents

There are a number of idents, both users and groups. All ident names have a MIMER_ prefix and group idents have a _GROUP suffix to aid with their identification.

The user ident MIMER_ADM can be thought of as an administrator, which is the reason why the user has been given the system privilege to create new idents and tables. Access and object privileges are not granted directly to the user but inherited through membership of MIMER_ADMIN_GROUP. This group of idents has a high level of privileges to manipulate the database tables and views but they do not have total uncontrolled access, only the MIMER_STORE ident has that. It should be possible to perform a lot of the course work from the MIMER_ADM ident.

The user MIMER_USR is granted membership of the MIMER_STORE_GROUP. As can be seen it is possible to grant a group membership of another group; in this case, the members of the MIMER_ADMIN_GROUP (now and in the future) are automatically granted whatever privileges are allocated to MIMER_STORE_GROUP. So when the execute privilege on the PSM procedure COMING_SOON is allocated to MIMER_STORE_GROUP it is automatically also granted to the members of MIMER_ADMIN_GROUP, which includes the user MIMER_ADM.

Note: At the end of the default installation of the example database, the permission to create further idents and databanks from example environment is revoked. This is since the default system is given public passwords, and therefore is open to any user. The revoke commands used are the following:

```
REVOKE DATABANK FROM MIMER_STORE;
REVOKE IDENT FROM MIMER_STORE CASCADE;
```

To deliberately open up the system, use the GRANT IDENT and GRANT DATABANK statements.

See the exdef.sql file for details.

The MIMER_STORE_MUSIC Schema

Schemas have been used to group objects related to a specific area; note that all the schemas are owned by the ident MIMER_STORE.

It is possible to ignore the schemas other than MIMER_STORE; they can be viewed as a black box for the purposes of an introduction to SQL.

A CREATE SCHEMA statement is used to demonstrate that, within the statement, an object doesn't have to be created before it can be referenced.

The DURATION domain introduces the INTERVAL data type.

Tables

Table name	Description
ARTISTS	Holds artist names (e.g. Bruce Springsteen) with a Soundex value based on the name. Shows that objects in schema other than the current have to be qualified with the owning schema name.
TITLES	Links an artist name with an item.
TRACKS	Holds track details, e.g. title, length.
SAMPLES	Holds samples from the tracks.

Views

View name	Description
DETAILS	This view includes the PRODUCT_DETAILS view and demonstrates a number of things: Tables can be included from more than one schema. The use of a correlation name in the PRODUCT_DETAILS table reference. Restriction conditions can be applied so that different users would see different results, in this case the ident MIMER_WEB would not see vinyl albums. One of the quirks in SQL – ITEM_ID has to be explicitly included in the select-list.
SEARCH	This view is based on DETAILS but includes additional selection restrictions. Note that the AS clause is noise and may be omitted but it does add clarity. This view demonstrates both forms.

Note that the CREATE SCHEMA statement includes grant object privilege statements as well as object definitions.

PSM Routines

The names in the PSM routines for this schema have been capitalized rather than the parts separated by an underscore.

PSM routine name	Description
`AddTitle`	Procedure to insert the base details for an album into the database, updating a number of tables. There are a number of checks on the input, for instance that the format is valid for the Music category and that the label exists in the `PRODUCERS` table – these introduce exception handlers. Uses another form of the row data type. Demonstrates the use of `CURRENT VALUE` of a sequence.
`AddTrack`	Procedure to insert the track details for an EAN code. Demonstrates the use of SQLSTATE values in an exception handler. Introduces the `RESIGNAL` statement and shows another option with `GET DIAGNOSTICS`.
`ArtistName`	Function to remove any leading definite or indefinite articles from a name.
`ArtistSearchCode`	Function to return the Soundex value of a character string after it has been processed by the `ArtistName` function.
`Search`	Result set procedure that searches the Music 'database' for matches based on the supplied title and artist details; a third parameter specifies the maximum number of rows to be returned (a value of zero suppresses this feature). Makes use of the `SEARCH` view. Includes the use of the `DISPLAY_ORDER` column in the `FORMATS` table to present the data in an order other than alphabetic. Each row is given a star rating that indicates the level of match (`****` = exact). The artist 'Bruce Springsteen' provides a number of matches. This procedure demonstrates the level of functionality that can be placed in the database.
`TitleDetails`	Result set procedure that returns music details for a given item identifier. Given the result from a search this would allow the user to 'drill down' into the displayed information. Illustrates how the compound statement label can be used to qualify a variable name. Shows that interval arithmetic can be performed.

PSM routine name	Description
TrackDetails	Result set procedure that returns any track details for a given item identifier. Usage is the same as for TitleDetails. Demonstrates the use of a user defined SQLSTATE.

Triggers

There are two statement triggers in the MIMER_STORE_MUSIC schema, both defined against the ARTISTS table. They are designed to ensure that the value of the ARTIST_SEARCH column is based on the artist name. The same technique as used in the MIMER_STORE schema to force an unconditional update is applied in the insert trigger.

The update trigger is written to show that it can contain the same level of functionality as any PSM routine; in this case it uses a cursor to process the updates.

Idents

Execute privilege on the PSM routines Search, TitleDetails and TrackDetails is granted to MIMER_STORE_GROUP.

Synonyms are created by MIMER_STORE for all tables and views in the MIMER_STORE_MUSIC schema. The synonyms are created with a MUSIC_ prefix for the DETAILS, SEARCH and TITLES tables/views.

The MIMER_STORE_BOOK Schema

This schema contains a table named TITLES, as does the MIMER_STORE_MUSIC schema.

Tables

Table name	Description
TITLES	Links an item with a list of authors and an ISBN. A book may have more than one author, the names are held as a list in the column AUTHORS_LIST in the form: surname, forenames; surname, forenames; …
AUTHORS	Links an item to an entry in the KEYWORDS table.
KEYWORDS	Holds each author in the form surname,[first initial]. An extension would be to categorize books and create a new table to form a link between an item and various categories, with the category being held in the KEYWORDS table.

Object privileges are granted on the MIMER_STORE_BOOK schema tables to MIMER_ADMIN_GROUP.

PSM Routines

PSM routine name	Description
VALIDATE_ISBN	Procedure to validate an ISBN. Uses an INOUT parameter. Shows how to CAST to an INTEGER and trap any error to validate that a string is numeric.
FORMAT_ISBN	Function to format an ISBN (e.g. insert hyphens to separate the country, group, publisher, title identifiers). Demonstrates the use of CASE statements, including where there is not an ELSE. Uses a row data type to simplify coding.
AUTHORS	Function to return the first author from a list of authors; if there is more than one author then a mark of omission is included.
AUTHORS_NAME	Function to format an author's name into surname[,initial].
KEYWORD_ID	Function to insert a word into the KEYWORDS table and return the identifier with which the keyword is associated.
CATALOGUE_AUTHORS	Given the list of authors associated with a book, extracts each author, calls the AUTHORS_NAME function and then the KEYWORD_ID function. Finally, creates a link between each name and the book in the AUTHORS table.
ADD_TITLE	Procedure to insert the base details for a book into the database, updating a number of tables. Inserts against the DETAILS view.
SEARCH	Result set procedure that searches the Book 'database' for matches based on the supplied title and author (surname, forename). The author 'Christie, Agatha' will provide a number of matches. Demonstrates a different method of searching.
TITLE_DETAILS	Result set procedure that returns book details for a given item identifier. Given the result from a search this would allow the user to 'drill down' into the displayed information.

Views

View name	Description
DETAILS	This view includes the PRODUCT_DETAILS view and is the equivalent of the DETAILS view in the MIMER_STORE_MUSIC schema.

Triggers

There are two statement triggers defined against the TITLES table. They are designed to maintain the entries in the AUTHORS and KEYWORDS tables. The update trigger is written to show how to use the OLD and NEW table aliases.

There is also an INSTEAD OF trigger defined on the DETAILS view. This is used by the ADD_TITLE procedure to update the underlying tables on which the view is based. Note that the INSTEAD OF trigger has to be defined before an insert statement against the view can be included, otherwise the join is not considered to be updateable.

Idents

Access privileges on the view DETAILS are granted to MIMER_ADMIN_GROUP.

Execute privilege on the PSM routines SEARCH and TITLE_DETAILS is granted to MIMER_STORE_GROUP.

Synonyms are created by MIMER_STORE for all tables in the MIMER_STORE_BOOK schema. The synonyms are created with a BOOK_ prefix for the DETAILS view and TITLES table.

The MIMER_STORE Schema Revisited

A quick return to the base schema to include two PSM routines that are outside the scope of an introduction to SQL.

PSM Routines

PSM routine name	Description
ORDER_ITEM	Procedure to associate an order for a quantity of a particular item against an order identifier.
BARCODE	Result set procedure that returns the book or music details for a given EAN. The intention would be to use this with a POS barcode reader.

The MIMER_STORE_WEB Schema

This schema provides some of the SQL functionality required to create a Web-based application to order items.

The ident-name in the AUTHORIZATION clause is currently restricted to be the name of the current ident.

The basic idea behind a Web application would be two tabs, one for Music and the other for Books. The relevant SEARCH routine provides a list of matches (one of the details returned is the ITEM_ID). The user would then have the ability to drill down to display further information (use MIMER_STORE_MUSIC.TitleDetails and MIMER_STORE_MUSIC.TrackDetails for music items and MIMER_STORE_BOOK.TITLE_DETAILS for book items.

If an item is selected for purchase (the quantity should be prompted for) and MIMER_STORE_WEB.ADD_TO_BASKET used to order the item (a blank SESSION_ID defines a new session). Once a session has been created the basket can be viewed using MIMER_STORE_WEB.VIEW_BASKET.

When an order has been completed the user needs to be identified by their e-mail and password (MIMER_STORE_WEB.VALIDATE_CUSTOMER) and then a call should be made to MIMER_STORE_WEB.PLACE_ORDER.

Tables

Table name	Description
SESSIONS	Maps an external session identifier with an internal order identifier. Keeps track of the date/time that the 'basket' was last accessed.

PSM Routines

PSM routine name	Description
SESSION_EXPIRATION_PERIOD	Function that returns an interval data type that defines the period in which a basket should be accessed.
DELETE_BASKET	Procedure to delete a specified basket session; alternatively a session of '*' will delete all 'baskets' that have expired. Note that the procedure deletes entries in the ORDERS table and relies on foreign key definitions in referencing tables to tidy up.
ORDER_ID	Function to return the order identifier associated with a specified session identifier. Raises an exception if the 'basket' hasn't been used within the period specified by SESSION_EXPIRATION_PERIOD.

PSM routine name	Description
VALIDATE_BASKET	Function that uses the ORDER_ID function to validate that the basket is still active. Uses an exception handler to catch any SESSION_INVALID exception raised by ORDER_ID. The exception handler will call DELETE_BASKET to remove a basket that has expired.
ADD_TO_BASKET	Function to place an order for a quantity of a specific item. If the session identifier is provided, then the order is placed against the relevant order identifier; if the session identifier is blank, then a new basket is created. The function returns the current session identifier.
VIEW_BASKET	Result set procedure that lists the items ordered. Uses a GROUP BY clause. Calls the BARCODE procedure, which is itself a result set procedure.
VALIDATE_CUSTOMER	Function to identify a customer by their e-mail address and password.
PLACE_ORDER	Procedure to order the contents of the basket. Procedure takes two in parameter: session identifier and customer identifier. Returns an order number, total price in euros, the local currency for the customer and the price in that local currency.

Triggers

There is a statement trigger that will fire after an update to the SESSIONS table. The trigger is designed to prevent any changes to the values of the SESSION_NO and ORDER_ID columns. This demonstrates that a trigger can be used to abort an SQL update operation.

Idents

A synonym is created by MIMER_STORE for the SESSIONS tables in the MIMER_STORE_WEB schema.

A new user ident, MIMER_WEB, is created to allow web-applications execute privilege on certain of the PSM routines.

Synonyms

A complete set of synonyms is created for the ident MIMER_ADM. This is to simply access to the tables in the classroom; the synonym can be used in place of the table-names that would have to be qualified with the schema-name.

Appendix C

Deprecated Features

This chapter discusses features and functionality that have been deprecated.

BSQL Commands

The following BSQL commands are deprecated but are supported for backward compatability.

LOAD

The `LOAD` command, used to load data from a sequential file into a target table, is now deprecated. Instead of `LOAD`, please see the *Mimer SQL System Management Handbook, Chapter 8, Loading and Unloading Data and Definitions*.

UNLOAD

The `UNLOAD` command, used to unload data from a table into a sequential file, is now deprecated. Instead of `UNLOAD`, please see the *Mimer SQL System Management Handbook, Chapter 8, Loading and Unloading Data and Definitions*.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Mimer SQL

System Management Handbook

Version 11.0

Mimer SQL, System Management Handbook, Version 11.0, December 2024
© Copyright Mimer Information Technology AB.

The contents of this manual may be printed in limited quantities for use at a Mimer SQL installation site. No parts of the manual may be reproduced for sale to a third party.
Information in this document is subject to change without notice. All registered names, product names and trademarks of other companies mentioned in this documentation are used for identification purposes only and are acknowledged as the property of the respective company. Companies, names and data used in examples herein are fictitious unless otherwise noted.

Produced and published by Mimer Information Technology AB, Uppsala, Sweden.

Mimer SQL Web Sites:
https://developer.mimer.com
https://www.mimer.com

Contents

Chapter 1

Introduction

Mimer SQL is an advanced relational database management system (RDBMS) developed by Mimer Information Technology AB.

Mimer SQL has a number of unique technical solutions to handle some of the more complicated functionality that a database management system must provide.

For example, Mimer SQL provides a solution to the problem of allowing simultaneous access to the database without the danger of a deadlock occurring. This greatly simplifies database management and allows truly scalable performance, even during heavy system load.

Another significant technical innovation is the data storage mechanism, which is constantly optimized for the highest possible performance and ensures that no manual reorganization of the database is ever needed.

Mimer SQL offers a uniquely scalable and portable solution, including multi-core support. The product is available on a wide range of platforms from small embedded and handheld devices running for example Android or Linux, to workgroup and enterprise servers running Linux, Windows, macOS and OpenVMS. This makes Mimer SQL ideally suited for open environments where interoperability, portability and small footprint are important.

The database management language Mimer SQL (Structured Query Language) is compatible in all essential features with the currently accepted SQL standards, see the *Mimer SQL Reference Manual, Chapter 3, Introduction to SQL Standards*, for details.

About this Manual

This manual describes how to establish, manage and maintain a Mimer SQL database. It is a general handbook for system administrators, describing in detail the various areas of responsibility and procedures to use when administering a Mimer SQL database system.

The information contained in this handbook generally applies to all the platforms supported by Mimer SQL.

From time to time platform-specific notes appear in the general description, presented as follows:

Linux:	Denotes information that applies specifically to Linux and macOS platforms.

VMS:	Denotes information that applies specifically to OpenVMS platforms.

Win:	Denotes information that applies specifically to Windows platforms.

There are also some appendices at the end of this handbook which contain information that applies to specific platforms.

Prerequisites

There are no prerequisites for users of this manual. However, it is advisable for the reader to be familiar with, and have a working knowledge of, the host computer operating system.

Related Mimer SQL Publications

- **Mimer SQL Reference Manual**

  contains a complete description of the syntax and usage of all statements in Mimer SQL and is a necessary complement to this manual.

- **Mimer SQL User's Manual**

  contains a description of the BSQL facilities. A user-oriented guide to the SQL statements is also included, which may provide help for less experienced users in formulating statements correctly (particularly the SELECT statement, which can be quite complex).

- **Mimer SQL Programmer's Manual**

  describes the usage of SQL embedded in application programs.

- **Mimer SQL Platform-specific documents**

  contain platform-specific information. A set of one or more documents is provided, where required, for each platform on which Mimer SQL is supplied.

- **Mimer SQL Release Notes**

  contain general and platform-specific information relating to the Mimer SQL release for which they are supplied.

Acronyms, Terms and Trademarks

Term	Description
API	Application Programming Interface.
BSQL	Facility for using SQL interactively or by running a command file.
Data source	ODBC term for a database.
Databank	A databank corresponds to a physical file in the host file system and is used to store tables. A databank may contain several tables.
Database	A Mimer SQL database consists of the system databanks and a number of user databanks.
Database home directory	The directory where the system databank file containing the data dictionary and some other files that form part of the database are located.

Term	Description
DCL	Digital Command Language.
Embedded SQL	The term used for SQL statements when they are embedded in a traditional host programming language.
JDBC	Database connectivity for Java.
MULTIDEFS	A file, on Linux and OpenVMS, containing parameters for controlling the database server for a Mimer SQL database.
ODBC	Open Database Connectivity, a specification for a database API in the C language, independent of any specific DBMS or operating system.
PSM	Persistent Stored Modules (i.e. Stored Procedures).
Shadow	A Mimer SQL databank may have one or more shadows. A shadow is a copy of the original (master) databank and is continuously updated by Mimer SQL.
SQL	Structured Query Language.
SQLHOSTS	A file, on Linux and OpenVMS, containing lookup information for all Mimer SQL databases accessible from the current node.
UNIX	UNIX is a trademark registered by the Open Group.
VMS	VMS is a trademark registered by Hewlett-Packard.
Windows	Windows is a trademark registered by Microsoft.

All other trademarks are the property of their respective holders.

System Management Responsibilities

Installation of a Mimer SQL system and the initial creation of the database environment is performed by a specially privileged operating system user, referred to as the system administrator.

In an established system, the system administrator is also responsible for the maintenance of the installation - system tuning, troubleshooting, backup/restore, and so on. The system administrator must have a good working knowledge of the host computer operating system.

SYSADM

In addition to system administration, there are certain management activities which are performed within the database, i.e. database administration.

A specially privileged Mimer SQL ident called `SYSADM` is created when a database is installed and this ident should be used whenever database administration activities are to be undertaken.

Note: The ident name `SYSADM` may not be changed.

SYSADM Privileges and Access Rights

Some of the Mimer SQL functionality requires privileges and access rights which are initially granted only to the SYSADM user, but which may be passed on to other Mimer SQL users.

SYSADM has SELECT access on the internal Mimer SQL data dictionary tables, permitting direct reading of the meta-data describing the system. In examining the contents of the data dictionary with the functionality provided, the user ident SYSADM has, by default, wider access rights than other users.

The user ident SYSADM does not, however, have general access to the contents of the database. Information stored in user-defined tables may only be accessed by other users if the creator of the table explicitly grants permission. In this context, SYSADM is treated just as any other database user.

SQL Statement Execution

At several places in this manual there are guidelines where SQL statements are shown. These statements can be executed from any ad-hoc SQL query tools, such as Mimer BSQL, DbVisualizer, etc.

Chapter 2

The Database Environment

This chapter describes the database environment which is composed of a set of Mimer SQL system databanks, one or more idents authorized to connect to the database and the databanks created by the idents. It also describes database security and data integrity.

The objects that are created in the database are described in the *Mimer SQL Reference Manual, Chapter 4, Mimer SQL Database Objects*.

The Data Dictionary

The database environment is controlled through a central data dictionary, stored in the system databank SYSDB and is automatically maintained by Mimer SQL. The data dictionary contains meta-data describing all the objects in the database. System access to the data dictionary tables is performed by internal routines and is transparent to the user.

Restricted facilities for examining the contents of the data dictionary are available to all users through the `LIST` and `DESCRIBE` functions in BSQL, see the *Mimer SQL User's Manual, Chapter 9, Mimer BSQL* for a more complete description of these facilities.

In general, a user may read data dictionary information for database objects to which they have access. The BSQL facilities use pre-defined views on the data dictionary tables to present the information in a structured form, see the *Mimer SQL Reference Manual, Chapter 13, Data Dictionary Views* for documentation on the data dictionary views available to all users.

The `SYSADM` user ident may read the contents of the data dictionary tables directly, and may grant SELECT access on the tables to any other user ident. The organization of the data dictionary tables is documented in *Appendix D Data Dictionary Tables* of this manual.

No individual user, including `SYSADM`, may update data dictionary tables directly. All write operations in the data dictionary are performed by internally controlled routines, to ensure consistency within the dictionary.

Idents

An ident in a Mimer SQL system is an authorized user of the system, or the collective identity of a group of users sharing common privileges.

There are three types of idents: USER, PROGRAM and GROUP idents.

USER Idents

USER idents are authorized to connect to a Mimer SQL database, by using the CONNECT statement in an application program or by entering the correct ident name and password in an interactive environment.

Any privileges a user ident holds may be exercised once the ident has logged on. USER idents are generally associated with specific physical individuals authorized to connect to the database.

An OS_USER login can be added to a user which allows the user currently logged in to the operating system to connect to a Mimer SQL database without providing a password. (If the Mimer USER ident name is the same as the operating system username, its possible to connect to Mimer SQL without providing username.)

If a USER with an OS_USER login is defined with a password in Mimer SQL, the ident may connect to Mimer SQL in the same way as any other user ident (i.e. by providing username and password).

PROGRAM Idents

PROGRAM idents may not initiate a connection to a Mimer SQL database, but may be entered from within an application program or interactive environment by using the ENTER statement.

A connection to the database should have been established before the ENTER statement is used. The ident using the ENTER statement must hold EXECUTE privilege on the PROGRAM ident.

Entering a PROGRAM ident is analogous to logging on as a USER ident, in that the PROGRAM ident gains access to the system and any privileges the ident holds become applicable.

PROGRAM idents are generally associated with specific functions within the system, not with physical individuals.

GROUP Idents

GROUP idents are collective identities for groups of USER or PROGRAM idents.

Any privileges granted to or revoked from a GROUP ident automatically apply to all members of the group.

Any ident can be a member of as many groups as required, and one group can include any number of members.

GROUP idents provide a facility for organizing the privilege structure in the database system. For examples showing the use of a GROUP ident, see the *Mimer SQL User's Manual*.

Idents – Access and Authority

USER and PROGRAM idents are authorized users of the system.

Every PROGRAM ident has a unique ident name and a private password which must be correctly supplied to the ENTER statement in application programs.

Every USER ident has a unique ident name and an optional private password which must be correctly supplied to the CONNECT statement in application programs. Alternatively a USER with an OS_USER login may access the database without explicitly providing a password on condition that the username for the user currently logged in to the operating system correspond to the definition of an OS_USER in the Mimer SQL database.

When Mimer SQL is installed, the user ident SYSADM, used for database administration, is automatically created. The password for SYSADM is defined when the system is installed, see *SDBGEN - Generating the System Databanks* on page *34*.

All idents in the system belong to a group which is specified by using the keyword PUBLIC in Mimer SQL statements. Privileges granted to PUBLIC are global to the system.

Schemas

A schema defines a local environment within which private database objects can be created. The ident creating the schema has the right to create objects in it and to drop objects from it.

When a USER or PROGRAM ident is created, a schema with the same name is created by default, and the created ident becomes the creator of the schema. This happens unless WITHOUT SCHEMA is specified in the CREATE IDENT statement.

When a private database object is created, its name can be specified in a fully qualified form which identifies the schema in which it is to be created. The names of objects must be unique within the schema to which they belong, according to the rules for the particular object-type.

If an unqualified name is specified for a private database object, a schema name equivalent to the name of the current ident is assumed.

Databanks

A Mimer SQL database consists of the Mimer SQL system databanks and a number of user databanks.

Each databank is one or several physical files in the host file system.

Mimer SQL System Databanks

The Mimer SQL system databanks are fundamental to the functioning of a Mimer SQL database and they are created during the process of installing a database.

System databanks are not used for storing user-defined information and cannot be updated directly by users.

If any one of the system databanks is damaged or missing, attempts to log on to Mimer SQL will fail. Backup and restore procedures for the system databanks are described in *Backing-up and Restoring Data* on page *67*.

The Mimer SQL system databanks are:

- SYSDB
- LOGDB
- TRANSDB
- SQLDB

SYSDB

SYSDB is the most important system databank as it stores the tables that make up the data dictionary, see *The Data Dictionary* on page *5*.

Among other things, the data dictionary holds information about the other databanks that make up the database, the tables each user databank contains, the users (idents) that are known to the Mimer SQL system and the access rights each ident has.

SYSDB is the only databank that must consist of only one file.

TRANSDB

TRANSDB stores information that is vital for keeping the database in a consistent state.

LOGDB

LOGDB records all write operations performed within transactions on SYSDB and user databanks which have been defined with the LOG option.

The information in this databank is used by the backup and restore facilities, see *Backing-up and Restoring Data* on page *67*, to restore the contents of a database in the event of a system failure.

A readlog facility is provided to allow the information in this databank to be examined, see *Mimer SQL User's Manual, Chapter 9, READLOG*.

SQLDB

SQLDB is used by the transaction handling mechanism to store row data read from the database, see *Transaction Control* on page *17*, and is used by Mimer SQL for temporary storage of result tables.

User Databanks

User databanks contain the tables in the database created by the users of the system. Typically these databanks are created by the system administrator and TABLE privilege is granted to the users of the system.

The CREATE DATABANK statement is used to create user databanks, see the *Mimer SQL Reference Manual, Chapter 12, SQL Statements*.

Except at the point when tables are created, the existence of databanks is transparent to users and application programs. When access is requested to a table, information in the data dictionary is used by Mimer SQL to locate the table and make it available, if permissible.

The division of a database into databanks is made on the basis of file handling considerations from the operating system viewpoint and on the basis of transaction control considerations from the database viewpoint. The use of databanks allows considerable flexibility in the physical placement of data on the computer system.

Databank Options

Databanks may be defined with the `LOG`, `TRANSACTION`, `WORK`, or `READ ONLY` options which determine transaction handling and logging behavior, as follows:

- **LOG**

  All operations on the databank are performed under transaction control. All transactions are logged.

- **TRANSACTION**

  All operations on the databank are performed under transaction control. No transactions are logged.

- **WORK**

  All operations on the databank are performed without transaction control (even if they are requested within a transaction), and are not logged.

- **READ ONLY**

  Only read only transactions are allowed. No transactions are logged.

- **TEMPORARY**

  This option is `SQLDB` specific.

Note: Operations performed on databanks with the `TRANSACTION` or `WORK` option cannot be restored in the event of system failure, see *Backing-up and Restoring Data* on page *67*.

If a databank is dropped from the database, the tables stored in the databank are also dropped and the physical databank files are deleted from disk.

Creating Idents and a Databank – an Example

In the following example, a databank and two users (idents) are created. The idents are then given the authority to create tables within the databank:

```
SQL> CREATE DATABANK DEVELOP SET FILE 'dev.dbf',
            FILESIZE 50 M, OPTION LOG;
SQL> CREATE IDENT DEVUSER AS USER USING 'devuser';
SQL> GRANT TABLE ON DEVELOP TO DEVUSER;
SQL> CREATE IDENT JIM AS USER;
SQL> ALTER IDENT JIM ADD OS_USER 'JIM';
SQL> ALTER IDENT JIM ADD OS_USER 'JAMES';
SQL> GRANT TABLE ON DEVELOP TO JIM;
```

Locating Databank Files

The system databank `SYSDB` is always stored in a file located in the directory defined as the home directory for the database, see *The Local Database* on page *28*. The file locations of all the other system databanks and the user databanks are stored in the data dictionary.

The file specification for the databank file is exactly as specified when the databank was created. If a databank file specification is given in full, it is unambiguously specified and no variable factors are involved in resolving the location of the file.

If a databank file specification appears in the data dictionary without an absolute directory name, the database home directory will be used to complete the file specification.

This substitution is applied whenever the location of the databank file must be determined, (i.e. when the databank is created or altered and whenever tables stored in it are accessed).

Subsequent redefinition of the database home directory or any variables used in the file specification will, therefore, alter the expected location of such databank files.

Linux: Databases on Linux platforms may be set up with a directory search path instead of a single home directory, see *The Local Database* on page *28*.

The first directory in the search path list must be the database home directory, where `SYSDB` is located. Other databank files can be located in any of the directories in the search path list.

VMS: Whenever a databank file is specified without a directory name under OpenVMS, it must be located in the database home directory.

If a logical name is included in the file specification, this will be recorded in the data dictionary and will be used whenever the location of the databank file is resolved.

Any logical names used in databank file specifications must be created as `GROUP` or `SYSTEM` wide logical names so that the database server process has access to them.

Win: Databases on Windows platforms may be set up with a directory search path instead of a single home directory, see *The Local Database* on page *28*.

The first directory in the search path list must be the database home directory, where `SYSDB` is located. Other databank files can be located in any of the directories in the search path list.

The flexibility achieved by not using full databank file specifications must be weighed against the loss of explicitly specified information from the data dictionary. In addition, the centralized use of mechanisms such as environmental variables or logical names in a complex system requires careful and disciplined management.

In particular, it is necessary for the database server process to have access to all relevant environmental variables and logical names in order to use them when accessing the databanks.

Organizing Databank Files

There are a number of factors involved in the organization of physical databank files that are important to database security and the overall performance of the Mimer SQL system.

Allocating Disk Space

Whenever possible, pre-allocate file space for databanks early in the lifetime of the databank file system.

The databank creation facilities allow the initial size of a new databank file to be specified in terms of the number of Mimer SQL pages. The size of a Mimer SQL page is 4 kilobytes.

The size of the databank file will be extended automatically by the database server during the lifetime of the databank as more space is required for data storage.

Linux: Under Linux, the environment variable `MIMER_EXTEND` can be set to the number of Mimer SQL pages by which all databank files will be extended. The default setting is 128.

VMS: By default, under OpenVMS, databank files will be extended by 1000 OpenVMS blocks at a time. The extend size for a databank file can be altered by using the following DCL command:

```
$ SET FILE/EXTENSION=extensionsize file.DBF
```

The databank file must not be in use by the database server (or accessed in single user mode), when this command is used.

Win: Under Windows, the number of Mimer SQL pages by which all databank files will be extended is determined by the Mimer SQL system and is not configurable.

An attempt to extend a file will fail if the disk is full, the databank attribute `MAXSIZE` is reached, or any imposed disk quota is exceeded.

Having a small file extension size may cause disk fragmentation leading to reduced I/O performance. In addition, if the databank is growing rapidly, the frequently occurring file extension operations may have a negative effect on performance.

A databank file which is created with the size it will actually need in production will be accessed more efficiently than one created with a small initial size and then incrementally extended.

The SQL statement `ALTER DATABANK SET FILESIZE` can be used to change the size of a databank file to a specified size. `ALTER DATABANK DROP FILESIZE` is used to shrink the database file as much as possible. The attributes `MAXSIZE`, `MINSIZE` and `GOALSIZE` can also be used to manage the databank file size. Refer to the *Mimer SQL Reference Manual, Chapter 12, ALTER DATABANK*, for details.

Mimer SQL databank files are organized internally into 4, 32 and 128 kilobyte databank blocks.

Accessing an internal databank block which is physically split over two or more distinct areas of allocated disk will require two disk read operations.

To avoid the risk of fragmenting the internal databank blocks, ensure that the number of disk blocks allocated for databank file extensions maps onto a whole number of 128 kilobyte databank blocks.

This will optimize disk I/O efficiency.

VMS: Disk blocks under OpenVMS are 512 bytes in size, therefore a disk cluster size which is a multiple of 8 will avoid fragmenting the 4 kilobyte databank blocks. The cluster size is set when formatting a disk.

Use the following command to check the cluster size of a disk that is already formatted:

```
$ SHOW DEVICE/FULL
```

Win: On Windows machines, disk clustering effects are hardware dependent and are not configurable.

Disks are typically configured in terms of an even number of 512 byte or 1024 byte disk blocks and will therefore always work efficiently with Mimer SQL databank files.

Use of disk defragmentation utilities may improve performance for large block I/Os.

Protecting Data Against Loss

For data security reasons, in case of a disk failure, it is strongly recommended that `LOGDB` is located on a disk unit that is physically separate from that on which the other databanks are located. See *Background Information* on page *67* for more information.

Ideally, `TRANSDB` and `LOGDB` should always be located on different physical disks which are served by separate disk controllers and no other databank files should be located on either disk.

The ordinary maintenance procedures for any computer system must involve backup and restore. A strategy, structure and procedure must be set up to include the Mimer SQL databases in the system backup routines. See *Backing-up and Restoring Data* on page *67* for a detailed discussion of backup and restore.

Note: A system without a complete and valid backup and restore procedure runs the risk of losing valuable data.

Balanced I/O

If several physical disk units are available, the various databanks should be distributed across the available disk units in order to balance the system I/O load.

To optimize the distribution of I/O across disks, place databanks on physical disks in such a way that databanks which are likely to be accessed at the same time are on different disk units.

It is generally the case that `TRANSDB` will be accessed at the same time as other databanks during a transaction.

Reserved Directories

The structure of the databank file system and procedures such as backup and restore are generally simplified if databank files are placed in directories reserved solely for that purpose. The system administrator should create and maintain a directory structure that best suits the local system.

It is very common practice to reserve entire disks for databanks to allow for the ultimate size of the files.

Other Performance Issues

The placement of databanks on physical disk units will depend on exactly how they will be used when the database system is in operation.

The following issues generally have a more significant effect on database performance than the disk I/O factors relating specifically to physical layout of the Mimer SQL database:

- the amount of virtual memory paging
- the speed of the disk
- the involvement of unnecessary network communication.

For example, to enhance performance, frequently accessed databanks such as TRANSDB may be placed on separate, high performance disks and sufficient memory should be allocated to avoid paging.

Altering Databank Locations

User databanks may be relocated by moving the physical file using operating system commands and then changing the file location stored in the data dictionary by using the `ALTER DATABANK` statement to specify the new file specification, see the *Mimer SQL Reference Manual* for the statement syntax.

The `ALTER DATABANK` statement may only be issued by the owner of the databank.

Example 1

1. Disconnect the databank from the system.

```
SQL> SET DATABANK databank_name OFFLINE;
```

2. Move or copy/delete the databank file to its new location.
3. Alter the databank filename in the data dictionary and reconnect the databank to the system.

```
SQL> ALTER DATABANK databank_name SET FILE 'new_filename';
SQL> SET DATABANK databank_name ONLINE PRESERVE LOG;
```

Example 2

By adding a new file and then deleting the original file, the databank tables will be available to users during the process.

1. Add a new file to the databank.

```
SQL> ALTER DATABANK databank_name ADD FILE 'new_file';
```

2. Drop the original databank file.

```
SQL> ALTER DATABANK databank_name DROP FILE 'old_file';
```

Facilities for changing the file specifications stored in the data dictionary for the system databanks, other than `SYSDB`, are provided by the BSQL program when a system databank is inaccessible, see *Re-creating TRANSDB, LOGDB and SQLDB* on page *77*.

`SYSDB` must always be located in the home directory for the database.

The location of a databank cannot be altered while the database server is accessing it or while it is being accessed in single-user mode.

Note: You cannot move databanks between databases by copying the databank file and using the facilities to alter the databank location recorded in the data dictionary. You must use the `LOAD` command. See *Chapter 8, Loading and Unloading Data and Definitions*.

Linux: Databases on Linux platforms may be set up with a directory search path instead of a single home directory, see *Locating Databank Files* on page *9*.

A databank created without specifying the directory in the file specification may be moved between any of the directories in the search path list without the need to alter anything in the data dictionary. Before being moved, the databank should be set offline to ensure that the file is not locked by the database server.

Accessing Databank Files

The databank files in a Mimer SQL database are accessed by the database server regardless of the user running the applications. The operating system privileges that apply to accessing the databank files are associated with the database server.

If a Mimer SQL database is accessed in single-user mode, see *Executing in Single-user Mode* on page *151*, the user must have the appropriate operating system level privileges in order to access the databank files.

Ownership of the databank files should not be confused with the creator of the databanks, which is internal to the Mimer SQL data dictionary. It is quite possible that a user who has created databanks is denied direct access at the operating system level to the files for those databanks.

Databank File Deletion

If a databank or shadow is dropped, the corresponding file will also be deleted from disk. Remember that dropping a Mimer SQL ident will also drop all objects, including databanks, that the ident has created.

When a databank is dropped, all shadows of the databank will also be dropped.

Note: If the databank is `OFFLINE` when it is dropped, the databank file (and any shadow files) will remain on disk in the file system and must be manually deleted.

Multifile Databanks

Tables and indexes reside in a databank. In the past a databank has been equivalent to a file in the file system. Now, a databank may consist of one or several files. If the files are placed on separate drives, both read and write performance is increased. This is because it is possible to both read and write to blocks in parallel on separate drives.

When there are several files, the database system will distribute the data evenly between the files automatically. In a b*-tree, blocks in one file may point to blocks in the same or other files in the databank in any fashion.

If a file is added to an existing databank, the file is added without moving any data. However, whenever new blocks are needed, the new file will be used and eventually the new file and the old one will hold equal amounts of data.

Any databank in the system can be a multifile databank except SYSDB. This means that the system databanks TRANSDB, LOGDB, and SQLDB may use the multifile support.

It is possible to add and drop files while a databank is in use. When a drop operation is done all the data in the dropped file is reallocated in the remaining files in the databank. This may take a long time if there is significant amount of data in the file. If the command is canceled, the operation is aborted and whatever data has been moved will remain in its new location. The file will in this case be active and new data will be stored in it.

Only one add or drop operation is allowed per databank concurrently. An error message, “databank locked” (error code -16172) is given if this is tried.

Tip: If you have a databank that you want to move to another location, for example, another disk drive this can be done by first doing ALTER DATABANK ADD FILE in the new location, followed by ALTER DATABANK DROP FILE for the old location. This will effectively move all the data in the databank to the new location/drive. Since both of these commands can be done on while the data is use, it can actually be done on a live system with no impact for applications except some extra I/O activity as the data is relocated.

Multifile databanks are very easy to both add and drop. However, there are some things to consider when using multifile databanks.

Since the files in a multifile databank have block references back and forth it means that all files in a multifile databank have a strong bond. It is, for example, not possible to replace one of the files. If this is done inadvertently the system will detect this and will give error message “One of the files for databank <%> does not match the other files in databank file set”, with error code -16264.

For multifile databanks there exist two sets of commands which may at first appear similar. They are:

```
ALTER DATABANK x ADD FILE 'filename'
```

and

```
ALTER DATABANK x ADD FILENAME 'filename'
```

The first command, ADD FILE, is used on a live system when the databank is accessible and we want to concurrently add a new file to the multifile set.

The second command, ADD FILENAME, is used when the data dictionary is out of date. If, for example, a backup has two files that we want to bring back. But if we have dropped one of the files since the backup, then in this case the data dictionary only has information about a single file. ADD FILENAME adds the file to the data dictionary.

So ADD FILE creates a new multi-file in a healthy system. ADD FILENAME is a data dictionary operation only, and is used to correct situations when we want to add existing files to a databank.

The corresponding DROP operations are also available.

```
ALTER DATABANK x DROP FILE 'filename'
```

and

```
ALTER DATABANK x DROP FILENAME 'filename'
```

The first command, DROP FILE, is used on a live system to migrate the data away from the file and then remove the file from the file system and the data dictionary.

The second command, DROP FILENAME, is used when the data dictionary is out of date. If, for example, a backup has a singe file that we want to bring back. But if we have added a file since the backup, then in this case the data dictionary has two files. DROP FILENAME removes the filename from the data dictionary.

So DROP FILE removes a file from a multifile databank in a healthy system. DROP FILENAME is a data dictionary operation only, and is used to correct situations when we want to remove references to files that no longer exists.

It can be noted that we want the data dictionary to have the correct number of files in the correct location. Each backup file may be placed in any of the locations pointed to by the data dictionary. I.e. copying file a1 to location /dev2/dbfiles/xx, and file b1 to location /dev1/dbs/yy works just as well as copying a1 to /dev1/dbs/yy, and b1 to /dev2/dbfiles/xx. I.e. the system will sort out the actual contents as long as the files belong together. (In this example, the data dictionary filenames are /dev1/dbs/yy and /dev2/dbfiles/xx for the databank.)

To be able to use the ALTER DATABANK commands the database server must be up and running. This means that all the system databanks must be available. Since the system databanks, except for SYSDB, may also consist of several files we must have a way to handle these. Any errors for the system databanks are handled with the batch SQL utility (bsql). When running bsql you will be guided along for each system databank in turn. It allows ALTER and DROP FILENAME operations to be performed. In addition, the databanks may be recreated from scratch or in some cases by reinitializing existing files. For these operations to be allowed you must log in as SYSADM. Some cases are covered in the later scenarios below.

Multifile scenarios

Here we describe a number of possible scenarios that may occur for multifile databanks.

Scenario 1

Let us assume you have a multifile databank with two files, each on its own disk. Let us also assume you have the same number of backup files.

One of the files is accidentally deleted. In this case you cannot bring back only the deleted file. You have to bring back BOTH files since they are strongly bonded. After both files are present the ALTER DATABANK x RESTORE USING LOG command is used to apply the changes since the backup. The restore will use both files automatically. RESTORE will only find data to restore if databank option LOG is used (see CREATE DATABANK or ALTER DATABANK command.)

Scenario 2

Let us assume you have a multifile databank with two files, each on its own disk. Let us also assume you have the same number of backup files.

One disk crashes and this disk is no longer accessible. Again, you have to copy BOTH of your backup files to the file system. One of the files overwrites the undamaged, remaining file, and the other file now needs to be placed in a new location. The SYSDB databank holds the data dictionary. The data dictionary points to the old location of the file on the device that is no longer available.

This location is now changed using the ALTER DATABANK x ALTER 'oldloc.dbf' SET FILE 'newlocation.dbf' command.

Next, the `ALTER DATABANK x RESTORE USING LOG` command is used to apply the changes since the backup.

Scenario 3

Let us assume you have a multifile databank with two files. You have taken an online backup of the databank. Online backups are currently always in a single file.

One of the files is accidentally deleted. In this case we want to bring back the backup. However, the data dictionary has two files and we only have a single file now. We remove one of the file references from the data dictionary with `ALTER DATABANK x DROP FILENAME 'filenametodel.dbf'`.

The dictionary only has a single file now for the databank. We can now copy the backup to the remaining file's location.

Next, the `ALTER DATABANK x RESTORE USING LOG` command is used to apply the changes since the backup.

Scenario 4

You have a multifile databank consisting of three files. You accidentally delete one the files and you have no backup of the databank.

In this case the entire contents of the databank is lost. There is no way to find the data without the missing file. If you have a databank with one file that is accidentally deleted the result is the same.

Scenario 5

Let us assume you have a multifile databank with two files. You have a backup of, among other files, the system databank `SYSDB`.

The file system with `SYSDB` is corrupted. You have to reinitialize it. You then bring in your backup of `SYSDB`. The `SYSDB` needs to be restored. This is done using the batch SQL utility. Batch SQL will prompt you for confirmation that you want to restore SYSDB (provided you have logged in as SYSADM).

In this case your `SYSDB` with its data dictionary will have the correct contents and should correspond to the other files present on the system.

Transaction Control

Transaction control provides a means of protecting the database from corruption which might arise from two users attempting to the change the same information at the same time and also provides the basis for ensuring database consistency, see *Database Consistency* on page *67*.

Optimistic Concurrency Control

Mimer SQL transaction management uses Optimistic Concurrency Control (OCC), which is described in the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security*.

This type of concurrency control overcomes many of the problems that can occur with conventional locking techniques (e.g. deadlocks and locks being retained by defunct connections). Superior performance is achieved because there is no need for the overhead of a deadlock detection mechanism, since deadlocks cannot occur.

Transaction Phases

A transaction is an atomic operation. Atomic means that all the changes that form the transaction are applied to the database, or none of them are applied.

Three transaction phases exist: build-up, during which the database operations are requested, prepare, during which the transaction is validated, and commitment, during which the operations performed in the transaction are written to disk.

The transaction begins by taking a snapshot of the database in a consistent state.

During build-up, changes requested to the contents of the database are kept in a write-set and are not visible to other users of the system. This allows the database to remain fully accessible to all users. The application program in which build-up occurs sees the database as though the changes had already been applied. Changes requested during transaction build-up become visible to other users when the transaction is successfully committed.

During build-up, a read-set records the state of the database as seen at the time of each operation (including intended changes). If the state of the database at commitment is inconsistent with the read-set, a conflict is reported and the transaction is rolled back (i.e. the write-set is erased and no changes are made to the database). This can happen if, for instance, a transaction asks to update a row which is deleted by another user after build-up has started but before the transaction is committed. The application program is responsible for taking appropriate action if a transaction conflict occurs.

Transaction control behavior in application programs and a number of the system facilities, notably Backup and Restore – see *Backing-up and Restoring Data* on page *67*, is controlled at the databank level by setting the option (`LOG`, `TRANSACTION`, `WORK` or `READ ONLY`) for the databank.

Only operations performed in databanks set up with the LOG option are logged in the Mimer SQL system databank LOGDB. Write operations against tables in `LOG` and `TRANSACTION` databanks must be performed under transaction control (i.e. within a transaction).

Refer to the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security* for more information.

Database Security

Mimer SQL supports a sophisticated system of access rights and privileges, which permit detailed control of database security.

The main components of the database security system are:

- idents
- system, object and access privileges
- restriction views.
- PSM routines

The Role of Idents in Database Security

Access to the Mimer SQL system as a whole is managed through the use of idents and privileges.

Careful advance planning of the hierarchical structure of idents in the database is vital to the long-term viability of the system. A poorly planned ident structure can easily become impossible to follow and control after a relatively short period of system use.

SYSADM

The Mimer SQL installation process creates one user ident, for use in database administration, with the name `SYSADM`.

The `SYSADM` ident has all the system privileges (`BACKUP`, `DATABANK`, `IDENT`, `SCHEMA`, `SHADOW` and `STATISTICS` – see *System, Object and Access Privileges* on page *20*, with the ability to grant these privileges to other idents, i.e. the privileges are held with the `WITH GRANT OPTION`.

The `SYSADM` ident also has `SELECT` access on all tables in the data dictionary, again, with the `WITH GRANT OPTION`. The `SYSADM` user is ultimately responsible for the structure of the whole system.

Re-creating system databanks can only be done by `SYSADM`, however, in other respects `SYSADM` is just an ordinary `USER` ident in the system.

It is quite possible, and may be advisable, especially in large systems, that `SYSADM` does not have access to the actual contents of the database; the database administration role should be concerned with objects in the system, not the actual data.

Public Group

All idents created in the system automatically belong to a logical group (specified using the keyword `PUBLIC` in Mimer SQL statements) which is intended to be used for granting global privileges.

Guidelines for Structuring Idents

The following general recommendations are made for structuring the idents in a system:

- Create `PROGRAM` idents for functional roles within the system. These are not coupled to any physical individual or group of individuals and thus have a lifetime independent of the turnover of personnel. (Database administration is an example of a functional role, but it is represented by a user ident rather than a program ident for practical purposes – see *Idents* on page *6* for details on idents).
- Create `USER` idents for physical users of the system. These may be dropped when the person concerned should no longer have access to the database. Do not grant privileges directly to user idents, other than membership to groups. Create the user idents `WITHOUT SCHEMA`. Administration is much simpler if privileges are granted through groups.
- Use `GROUP` idents to represent logical classes of users in the system. Grant privileges to groups rather than to individuals. This makes the granting of access rights to the system easier to organize and a clearer overview of the privilege structure within the system is maintained. It also means that new idents can be granted suitable privileges efficiently through membership in one or more groups.

- Grant the privilege to create objects (DATABANK, IDENT and TABLE privileges) to program idents only. In this way, individual USER idents may be dropped with no cascade effects (see *Cascade Effects Between Privileges* on page *22*). (Creation of domains requires no special privilege and may thus be performed by any ident with a schema. Creation of views requires only SELECT access to the table on which the view is based).
- Use the WITH GRANT OPTION sparingly and try to minimize the number of levels in the ident hierarchy. This reduces the risk of cascading revocation of privileges, see *Cascade Effects Between Privileges* on page *22*.

If these recommendations are followed, the maintenance of the ident structure in the system will be much more straightforward. Access to the contents of the database will be granted to relatively few GROUP idents instead of many individual program or user idents.

When a physical individual should no longer have access to the database, the corresponding USER ident can be dropped with no cascade effects.

System, Object and Access Privileges

Each ident is given privileges within the system which determine the operations the ident is permitted to perform.

Note: In addition to holding any relevant privilege(s), an ident must also be the creator of at least one schema before the ident is able to create private database objects (i.e. domains, functions, indexes, modules, procedures, sequences, synonyms, tables, triggers, types and views) - see *Schemas* on page *7*.

Privileges may be granted either directly or by making the ident a member of a GROUP ident. The privileges are classified as follows:

System Privileges

System privileges give the right to create global objects in the database. There are the following system privileges:

System Privilege	Description
BACKUP	gives the right to perform backup and restore operations
DATABANK	gives the right to create databanks
IDENT	gives the right to create idents and schemas
SCHEMA	gives the right to create schemas
SHADOW	gives the right to create shadows and perform shadow control operations
STATISTICS	gives the right to execute the UPDATE STATISTICS statement.

System privileges are granted to SYSADM at installation time and may be passed on to other idents with or without the WITH GRANT OPTION.

An ident receiving a privilege with the WITH GRANT OPTION may pass the privilege on to another ident.

Object Privileges

Object privileges give rights associated with certain specified objects in the system. There are the following object privileges:

Object Privilege	Description
TABLE	gives the right to create tables in a given databank
SEQUENCE	gives the right to create sequences in a given databank
EXECUTE	gives the right to execute a function, procedure or per-compiled statement, or the right to enter (become) a specified program ident
MEMBER	makes an ident a member in the specified GROUP
USAGE	gives the right to specify the nameddomain where a data type would normally be specified (in contexts where use of a domain is allowed), or the right to use a specified sequence or collation.

Object privileges are initially, automatically, granted only to the creator of the object (e.g. the creator of a databank automatically has TABLE privilege on the databank).

The privileges may be passed on to other idents with or without the WITH GRANT OPTION.

Access Privileges

Access privileges give rights of access to the contents of a specified table or view. There are the following access privileges:

Access Privilege	Description
SELECT	gives the right to read the table contents
INSERT	gives the right to add new rows to the table (this privilege may be limited to specified columns within the table)
DELETE	gives the right to remove rows from the table
UPDATE	gives the right to change the contents of existing rows in the table (this privilege may be limited to specified columns within the table)
REFERENCES	gives the right to use the primary or alternate keys of the table as a foreign key from another table (this privilege may be limited to specified columns within the table).

In addition to the five access privileges listed above, the keyword ALL may be used as a shorthand method of specifying all the privileges possessed by the granting ident. For example, if an ident has only SELECT and UPDATE privileges on a table and ALL is granted on that table to a new ident, the new ident will only be given SELECT and UPDATE.

Access privileges are initially granted to the creator of the table with the WITH GRANT OPTION. The privileges may be passed on to other idents with or without the WITH GRANT OPTION.

Certain operations are not controlled by explicit privileges, but may only be performed by the creator of the object involved. These operations include ALTER (with the exception of ALTER IDENT, which may be performed by either the ident itself or by the creator of the ident), DROP and COMMENT. Privileges may only be explicitly revoked by their grantor, however cascade effects may go wider.

Cascade Effects Between Privileges

Dropping an object from the database or revoking a privilege from an ident may have cascade effects on other objects and idents, depending on the way the database is organized.

The keywords CASCADE and RESTRICT may be used in the DROP and REVOKE statements.

When using RESTRICT (the default), the operation will fail with no changes being made if any cascade effects result from it.

When using CASCADE, the following operations have the consequences described:

- If an ident is dropped, all objects created by the ident are dropped and all privileges granted by the ident are revoked.
- If a databank is dropped, all tables in the databank are also dropped.
- If a table is dropped, all views and synonyms based on the table are dropped. Also, triggers and routines that references the table are dropped.
- If a privilege with the WITH GRANT OPTION is revoked from an ident, all instances of that privilege granted to other idents under the authorization of that WITH GRANT OPTION are also revoked. The WITH GRANT OPTION can be revoked separately.
- If SELECT privilege on a table is revoked from an ident, views created by the ident under the authorization of that SELECT privilege are dropped.

If DATABANK privilege is revoked from an ident, existing databanks created under that privilege are not dropped.

The cascade effects of revoking privileges only occur when the last instance of the privilege is revoked (a new instance of the privilege is created each time the privilege is granted to the same ident on the same object). An ident grants privileges, creates views and so on under the authorization of the most recently received valid instance of the WITH GRANT OPTION, SELECT or other relevant privilege.

The data dictionary keeps a record of the specific instance of an authorization under which an operation was performed. The cascade effects apply only to privileges granted or objects created under the specific instance of the authorization which is being revoked.

This is illustrated in the example cases that follow:

CASE 1

1 A grants with grant option to M

M grants to X

2 B grants with grant option to M

M grants to Y

3 A revokes from M

Both X and Y keep privileges

4 B revokes from M

Both X and Y lose privileges

CASE 2

1 A grants with grant option to M

2 B grants without grant option to M

M grants to X

M grants to Y

3 A revokes from M

M loses grant option

Both X and Y lose privileges

4 B revokes from M

M loses privilege

Restriction Views

Views are a powerful tool for restricting user access to specific parts of the database and they complement the use of access privileges in maintaining database security.

By defining restriction views (i.e. views based on one table but restricted only to specific rows and/or columns in the table), access may be provided to a subset of the contents of a table without affecting the physical database structure. In this way, the database may be designed optimally according to the relational model, while user access can be defined according to actual data retrieval requirements.

Data Integrity

The following facilities are available for ensuring the integrity of a Mimer SQL database:

- domains
- entity integrity (non-null primary keys)
- referential integrity (foreign keys)
- table integrity
- view integrity

Domains

Domains define sets of permissible values. By assigning a table column to a domain when the table is created or altered, the values which the column may contain are restricted to those defined in the domain. Any number of columns may use a given domain.

The ident defining a table column must hold `USAGE` privilege on any domain used.

A default value may also be defined for a domain. The domain default value is inserted into a column defined using the domain when data is inserted without an explicit column value being specified.

If the default value for the domain is defined outside the range of restriction values for the column, attempts to store the default value in a column using the domain will fail. In such a case an explicit value must always be specified when inserting data into the column.

The use of domains in table definitions is recommended, since this can provide an automatic check on the validity of data inserted into the column. However, domain definitions should be carefully planned, since a domain definition cannot currently be altered after it has been defined.

Entity Integrity

Entity integrity refers to the requirement that every row in a table must be uniquely identified and that no row in a table may be identified by null (i.e. by an unknown value).

Entity integrity can only be enforced if a primary key constraint or unique constraints are applied.

All primary key columns in tables created by Mimer SQL are defined as `NOT NULL`, thus ensuring entity integrity. Other (i.e. non-primary key) columns may also be defined as `NOT NULL` as required.

Referential Integrity

Referential integrity refers to the requirement that data entered into a table in the database must already be present in another table (e.g. a component may not be entered in a parts list if it does not already exist in the set of known components in the database).

Mimer SQL supports referential integrity through the `FOREIGN KEY` clause in the `CREATE TABLE` statement.

The properties of a `FOREIGN KEY` are as follows:

- The columns defined as a foreign key must correspond exactly in number and data type to the primary key or unique key columns in the referenced table.
- Data inserted into the foreign key columns (by either `INSERT` or `UPDATE` operations) must either already be present in the primary key or a unique key of the reference table or include at least one null column.
- A primary or unique key value that is referenced by a foreign key must not be removed by an update operation. It may be possible to remove such a value with a delete operation provided the `ON DELETE` rule is used to update the referencing table in a way that preserves the referential integrity.

Note: Referential integrity constraints are effectively checked at the end of the `INSERT`, `DELETE` or `UPDATE` statement. Both the table containing the foreign key reference and the referenced table must be stored in a databank with either the `TRANSACTION` or `LOG` option.

Table Integrity

Table integrity refers to the facility in Mimer SQL of defining `CHECK` clauses in a table definition, whereby the contents of one column is checked against the contents of one or more other columns in the same row of the table.

Data may only be entered into the table if the `CHECK` constraint is not violated.

View Integrity

View integrity refers to the facility in Mimer SQL of including a `WITH CHECK OPTION` clause in a view definition. If a view is defined with a `WITH CHECK OPTION`, data which violates the definition of the view may not be entered into the view by `INSERT` or `UPDATE` operations.

When a view is defined with a `CHECK OPTION`, any views defined on that view will inherit the `CHECK OPTION`.

Chapter 3

Creating a Mimer SQL Database

Once the Mimer SQL software is installed, the database environment must be created. This involves the following activities:

- registering the name of the database on each node in the network from which it is to be accessed.
- generating the Mimer SQL system databanks `SYSDB`, `TRANSDB`, `LOGDB` and `SQLDB` as well as the database administration ident called `SYSADM`.
- creating idents and data objects in the database using the data definition statements in Mimer SQL.

Linux: On Linux and macOS the provided `dbinstall` command performs all necessary installation steps to create an initial database and getting it up and running. The options available in `dbinstall` give opportunities to control and carry out the following:

- Deciding a database home directory
- Registering the database
- Deciding the SYSADM password
- Creating the system databanks, including the data dictionary
- Deciding owner of the database
- Setting up the networking environment
- Setting up autostart procedure
- Setting up a data source definition for ODBC use
- Creating an example database
- Creating a basic development setup with an OS_USER
- Creating the default database configuration file
- Starting the database created

Many of these tasks are described in a more general and detailed manner further on in this chapter.

Note: On macOS the `dbinstall` command is complemented with a Graphical User Interface.

3.1 Registering the Database

In a network environment, the name of a database must be registered on each node from which it is to be accessed.

A database is created as a local database on the node where it resides and it is defined as a remote database on each other node in the network from which access to it is required.

Linux & VMS: On Linux, macOS and OpenVMS nodes, the name of a database is registered by creating an entry for it in the `SQLHOSTS` file. See *The SQLHOSTS File on VMS and Linux* on page *155* for details about this file.

All users must have read access to the `SQLHOSTS` file on the machine they are using in order to run applications and utilities accessing Mimer SQL databases.

Win: On a Windows node, the name of a database is registered by running the Mimer Administrator. The Mimer Administrator adds information about Mimer SQL databases to the Windows registry. Refer to the Windows help provided with the Mimer Administrator for details on how to use it.

3.1.1 Database name recommendations

When selecting the database name there are some guidelines to be considered. The most recommended names are those built on ordinary letters, possibly combined with digits and dash/underscore. The case of letters used is not significant.

Please regard the following:

- Avoid having a database name that contains digits only. A database named as a number can get syntactical problems when used in commands, for example where optional numerical parameters can be given.
- Avoid using question marks and asterisks, since they can be treated as wildcard characters in some situations.
- Avoid using white space in the name, since this will make the use of the name more complex, requiring use of quotes or escape characters.
- Avoid using quotes in the database name, since this will be confusing and hard to use.

3.2 The Local Database

A local database is one that resides on the machine where its database server executes (i.e. the system databank file containing the data dictionary exists on a local disk).

A local database definition registers the database by specifying a name (which is not case sensitive) and a home directory for the database.

It also involves specifying various parameters which configure the database server that is started for the database.

Linux & VMS:	The definition of a local database under Linux and OpenVMS involves specifying the database name and the database home directory in the `SQLHOSTS` file, see *The SQLHOSTS File on VMS and Linux* on page *155*. Parameters that control the database server are specified in the `MULTIDEFS` file which is located in the database home directory, see *The MULTIDEFS File on VMS and Linux* on page *161* for details about this file and the parameters it contains. The `MULTIDEFS` file is automatically created with appropriate default values for all parameters when the database server is first started.

Win:	The definition of a local database under Windows is created by running the Mimer Administrator and specifying the required parameters, including those that control the database server, which are stored in the Windows registry. Windows help is provided with the Mimer Administrator to guide you through the creation of a local database. Default values are supplied for all parameters except the database name and home directory.

A fully created local database, complete with its Mimer SQL license key, see *Mimer SQL License Key* on page *30*, Mimer SQL system databanks, see *SDBGEN - Generating the System Databanks* on page *34*, user databanks and the ident and data structure contained in them, see *Establishing the Ident and Data Structure* on page *36*, constitutes a Mimer SQL database.

A completely created local database can only be accessed from the machine on which it resides. If the database is to be accessed from a remote node, connected to the local machine via a network connection, a remote database definition for the database must be created on the remote node.

3.3 Accessing a Database Remotely

In order to access a database that resides on another network node, the database must be created as a local database on the node it resides on and a remote database definition must be set up on the node from which the database is to be accessed.

The purpose of the remote database definition is to define a link with a database that resides elsewhere on the network. The name used for the remote database definition must be the same as that given to the local database it represents.

The definition for the remote database contains the communication parameters required for accessing the database over the network.

Linux: The definition of a remote database under Linux involves creating an entry in the `/etc/sqlhosts` file, see *The SQLHOSTS File on VMS and Linux* on page *155* for information on the parameters involved.

On the database server computer, the mimer service should be defined in `/etc/services` and a port dispatcher should be defined in `/etc/inetd.conf`.

This is usually done automatically by the dbinstall command. For details, see the Linux Getting Started G.

VMS: The definition of a remote database under OpenVMS involves creating an entry in the `SQLHOSTS` file, see *The SQLHOSTS File on VMS and Linux* on page *155* for information on the parameters involved.

Win: The definition of a remote database under Windows is set up by running the Mimer Administrator and specifying the required parameters. Windows help is provided with the Mimer Administrator to guide you through the creation of a remote database.

3.3.1 Client/Server Interface

Once the remote database definition has been set up and provided that the Mimer SQL client/server communications have been established correctly, access to a database that resides on a remote machine is performed transparently.

The Mimer SQL client/server communications interface is integrated into the database server. The database server process manages all connections.

All Mimer SQL applications may use the client/server interface without having to make any special provision in the application code. The client/server interface is automatically activated whenever a remote database is targeted.

The Mimer SQL client/server protocol is identical on all Mimer SQL platforms. This means that a Mimer SQL client on any machine type may access a Mimer SQL server for a remote database on any of the platforms on which Mimer SQL is implemented.

3.4 Mimer SQL License Key

To start the database server and to establish connections to the database, a license key is required. (A key valid for development and evaluation only is included in the Mimer SQL distribution.)

Whenever a user connects to a Mimer SQL database, the computer identification and the license key will be checked by the database server to determine access rights. If access is denied, the connect attempt will be aborted and an error message will be shown.

The Mimer SQL license key contains the following (encrypted) information:

- The maximum number of users that may use the database servers running on the same computer node at any one time.
- The maximum number of network users that may use the database servers running on the same computer node at any one time.
- The node name of the computer (in the case of a specific key) or a lifeboat key which is valid for any computer of the platform type for which it was issued (e.g. any Linux machine).
- Version number.
- Expiration date for the key.

The key data is case insensitive and space characters are ignored.

Linux & VMS: The `mimlicense` application is used to administrate the license key file. See *MIMLICENSE - Managing the license key* on page *32* for information on how to use MIMLICENSE.

Win: The Mimer Administrator is used to enter the Mimer SQL license key. The key is distributed in a `.mcfg` file.

When you double-click on a `.mcfg` file, the Mimer Administrator is automatically invoked to install the key.

Refer to the Windows help provided with the Mimer Administrator for details on how to manually enter the Mimer SQL license key.

The Mimer SQL license key is provided by your Mimer SQL distributor.

In order to be able to generate the key, your Mimer SQL distributor must know the node name, or serial number (depending on platform) of the computer on which the database server will run. On non-Windows platform, use `mimlicense` to retrieve this information:

```
mimlicense --cpuid
```

Or, VMS-style:

```
MIMLICENSE/CPUID
```

Win: When the dialog box which is used to enter a Mimer SQL license key is opened in the Mimer Administrator, the node name of the computer will be displayed.

Refer to the Windows help provided with the Mimer Administrator for details on how to open the dialog box.

When the number of Mimer SQL users is increased or new Mimer SQL functionality is added to the site, a new Mimer SQL license key will be provided.

The Mimer SQL license key uses the node name of the computer to link Mimer SQL to the computer it is authorized to run on. This allows for hardware replacement in the event of a failure in the computer system. If a replacement computer is given the same node name as the one it is replacing, the Mimer SQL license key remains valid for the new hardware.

3.5 MIMLICENSE - Managing the license key

Linux & VMS: The `mimlicense` application is currently available on Linux, macOS and OpenVMS.

Win: On Windows, the Mimer Administrator is used to administrate the license keys.

The `mimlicense` application is used to administrate the license key file. Keys may be added, removed or updated by using `mimlicense`. `mimlicense` may also be used to list and describe the contents of the key file.

Note that the database server must be restarted for the key changes to apply.

3.5.1 Syntax

The `mimlicense` program is controlled by flagged information specified on the command-line.

The overall syntax (expressed in long form Unix-style) is as follows:

```
mimlicense [-a hexcode | -c | -d keyid | -f file | -l | | -r | -i |-c | -n]

mimlicense [--add=hexcode | --combined | --delete=keyid | --filename=file |
    --cpuid | --list | | --nologo | --remove]
mimlicense [-v|--version] | [-?|--help]
```

3.5.2 Command-line Arguments

Unix-style	VMS-style	Function
`-a hexcode` `--add=hexcode`	`/ADD=hexcode`	Add a license key.
`-c` `--combined`	`/COMBINED`	Describe what the combined keys permits.
`-d keyid` `--delete=keyid`	`/DELETE=keyid`	Delete the specified key.
`-f filename` `--file=filename`	`/FILE=filename`	Add a license key from a `.mcfg` file.
`-i` `--cpuid`	`/CPUID`	Show the computer's CPU id.
`-l` `--list`	`/LIST`	List the contents of the key file.
`-n` `--nologo`	`/NOLOGO`	Silent mode, i.e. execution with no output.
`-r` `--remove`	`/REMOVE`	Remove keys. `mimlicense` will prompt for each key if it is supposed to be removed or not.

Unix-style	VMS-style	Function
`-v` `--version`	`/VERSION`	Show `mimlicense` version information.
`-?` `--help`	`/HELP`	Show help text.

Linux: The Unix-style command-line flags must be used on a Linux or macOS machine. Both short form arguments (e.g. `-r`) and long form arguments (e.g. `--remove`) are supported.

The name for the key file is:

```
/etc/mimerkey
```

VMS: Either the Unix-style or the VMS-style command-line flags may be used on an OpenVMS machine – see the *Mimer SQL VMS Guide* for more details.

The name for the key file is:

```
SYS$SPECIFIC:[SYSMGR]MIMERKEY.DAT
```

3.6 SDBGEN - Generating the System Databanks

The Mimer SQL system databanks SYSDB, TRANSDB, LOGDB and SQLDB are generated by the SDBGEN program.

SDBGEN will load the system tables, see *Data Dictionary Tables* on page *175*, and defines the data dictionary views detailed in the *Mimer SQL Reference Manual*.

Note: A databank created for one SYSDB cannot be accessed by using a different SYSDB even if identical data dictionary definitions are created in it.

Linux: A local database is set up on a Linux or macOS node by running the dbinstall command (see the dbinstall man-page) and SDBGEN is started automatically when required.

The databank files are by default created in the database home directory which is not the ideal arrangement from a security and performance point of view, see *Re-creating TRANSDB, LOGDB and SQLDB* on page *77* for guidelines relating to placement and organization.

If there is more than one disk available on the system, it is recommended that directories be created on those disks specifically for locating databanks. When created, the LOCAL definition for the database should be updated in the /etc/sqlhosts file by changing the single home directory path to a directory path list that includes these directories. The list is colon separated as can be seen in the following example, where the database is called “hotel”:

```
hotel /usr/db/hotel:/extra/db/hotel:/extra2/db/hotel
```

After this update is made, the database server can be stopped and the selected databank files can be moved from the database home directory to their new locations. When moved, the database server can be started again.

VMS: In order to create the Mimer SQL system databanks for a local database on an OpenVMS node, an entry for the database must be specified in the SQLHOSTS file. SDBGEN should then be executed to create the actual database. See *Syntax* on page *35* for information on how to run SDBGEN.

Win: You set up a local database on a Windows node by running the Mimer Administrator. The Mimer Administrator invokes the SDBGEN program in order to create the system databanks when required.

The system databanks are distributed automatically, depending on the number of disks available. Windows help is provided with both the Mimer Administrator and SDBGEN to guide you through setting up a local database.

3.6.1 Setting the Initial Size

The initial size for each of the Mimer SQL system databanks can be specified.

The size for the databanks is specified in Mimer SQL pages. The size of a Mimer SQL page is 4 kilobytes.

3.6.2 Setting Password for System Administrator

The database administration ident `SYSADM` is also created and a password (passwords are case-sensitive) must be specified for this ident. It should be chosen carefully and changed at appropriate intervals the `ALTER IDENT` statement.

Caution:	Care should be taken to safeguard the `SYSADM` password, because if it is lost it cannot be retrieved from the system and it is not possible to set a new one.

3.6.3 Syntax

Linux:	On Linux or macOS, the `dbinstall` utility is used to create the system databanks.

Win:	On Windows, the Mimer Administrator is used to create the system databanks.

The `SDBGEN` command has two purposes. Either to create a new set of system databank files, or to upgrade database files created in an earlier version of Mimer SQL to the current version.

The `SDBGEN` program is controlled by flagged information specified on the command-line.

The syntax (expressed in Unix-style) for creating databank files is as follows:

```
sdbgen [-p pass] [database [syssz [trafn [trasz [logfn [logsz [sqlfn
    [sqlsz]]]]]]]]

sdbgen [--password=pass] [database [syssz [trafn [trasz [logfn [logsz
    [sqlfn [sqlsz]]]]]]]]

sdbgen -u|--upgrade [database]

sdbgen [-v|--version] | [-?|--help]
```

3.6.4 Command-line Arguments

When creating databanks files:

Unix-style	VMS-style	Function
`-p password` `--password=password`	`/PASSWORD=password`	Password for `SYSADM`
`-u` `--upgrade`	`/UPGRADE`	Initiate a database upgrade.
`-v` `--version`	`/VERSION`	Display version information.
`-?` `--help`	`/HELP`	Display help text.
`database`	`database`	Database name
`syssz`	`syssz`	Size of `SYSDB`, 4K pages (prompted for if omitted.)
`tfn`	`tfn`	Filename for `TRANSDB`
`tsz`	`tsz`	Size of `TRANSDB`, 4K pages (prompted for if omitted.)
`lfn`	`lfn`	Filename for `LOGDB`
`lsz`	`lsz`	Size of `LOGDB`, 4K pages (prompted for if omitted.)
`sfn`	`sfn`	Filename for `SQLDB`
`ssz`	`ssz`	Size of `SQLDB`, 4K pages (prompted for if omitted.)

If the password parameter is omitted, the `SDBGEN` command will prompt for all parameters that are missing, including the password for the `SYSADM` user.

If the password parameter is given, the `SDBGEN` command will not prompt for any missing parameters, but use default values.

If the `database` parameter is missing, the environment variable `MIMER_DATABASE` is used to determine which database the databank files should be created for.

3.7 Establishing the Ident and Data Structure

Once the local database environment has been created for a Mimer SQL database (database name, server parameters, system databanks, the `SYSADM` ident plus the system tables and views), the data structure for the database (idents, user databanks, tables, and so on) can be created using Mimer SQL data definition statements.

Mimer BSQL allows the execution of a sequential file which can then be used as a permanent record of the `CREATE` statements used to create the database objects, see the *Mimer SQL User's Manual, Chapter 9, Mimer BSQL*.

Mimer BSQL also supports the saving of input and/or output to a log file (using the LOG command), so this facility could be used to create a permanent record of an interactive Mimer BSQL session which could be run again at a later date. Mimer BSQL, however, only has limited support for error handling.

An application program using embedded SQL (ESQL), JDBC or ODBC can also be used, but this requires more work on the part of the programmer and it provides a less concise record of the ident and data structure in the database.

Third party SQL tools are also available which may be used to create the database data structure.

Caution: A sequential file intended for non-interactive execution in Mimer BSQL can include username and password information relating to any CONNECT statements used. For security reasons, such a file should be well protected in the operating system, preferably with any username and password edited out of any permanent copy of the file.

An example database is delivered with Mimer SQL. See *Mimer SQL User's Manual, Chapter B, The Example Environment*.

3.8 Managing Database Connections

This section describes how users connect to a database and how several simultaneous connections from an application can be handled.

The following SQL statements are used for connection management:

- CONNECT
- DISCONNECT
- SET CONNECTION

See the *Mimer SQL Reference Manual, Chapter 12, SQL Statements* for details.

3.8.1 Selecting a Database

Applications establish database connections with the CONNECT statement, which specifies the database by name.

An application may connect to any of the databases which have been made accessible from the node where the application is running, see *Registering the Database* on page *28*. Some applications which are part of the Mimer SQL distribution allow the database name to be specified as a command-line argument.

The database may be located on the same machine as the application program (a local database), or on a remote machine accessed over a network (a remote database). The network connection is handled by the Mimer SQL software and this is completely transparent to the application program, see *Client/Server Interface* on page *30*.

A database is normally accessed by one or more users via the database server. It is also possible for one user to access a local database directly in single-user mode, provided the database server for it is not running and the operating system user has the appropriate access rights to the database files, see *Executing in Single-user Mode* on page *151*.

3.8.1.1 The Default Database

The default database will be used if the CONNECT TO DEFAULT statement is used, or if the database name in the CONNECT statement is specified as an empty string.

The default database can be any of the local or remote databases that are accessible from the node the application program is running on.

The database that is actually selected by a default connection depends on whether a node-specific or user-specific default database is defined at the time the connection is attempted.

Programs supplied as part of the Mimer SQL distribution (e.g. Mimer BSQL) will use the default database when database is not specified on the command line.

3.8.1.2 Defining a Node-specific Default Database

One default database can be defined for each node in a network.

Linux & VMS: The default database for Linux, macOS and OpenVMS nodes is defined by specifying the name of the database in the `DEFAULT` section of the `SQLHOSTS` file, see *The SQLHOSTS File on VMS and Linux* on page *155*.

Win: The default database for a Windows node is defined by using the Mimer Administrator to create a System Wide Mimer ODBC Data Source with the name `default` and associating it with the selected database.

Refer to the Windows help provided with the Mimer Administrator for details on how to create System Wide Mimer ODBC Data Sources.

3.8.1.3 Defining a User-specific Default Database

There may be times when an individual user may wish to override the default database defined for the local machine. This is done by defining a user-specific default database, which will be chosen in preference to the node-specific one.

Linux & VMS: A user-specific default database is defined under Linux, macOS and OpenVMS by setting the environment variable or logical name called `MIMER_DATABASE` to be the name of the required local or remote database, as stated in the `SQLHOSTS` file.

If the `MIMER_DATABASE` variable is set, all default connections will be made to the database it identifies.

If the `MIMER_DATABASE` variable is not set, default connections will be made to the node-specific default database for the local machine.

Win: A user-specific default database is defined under Windows by using the Mimer Administrator to create a User Specific Mimer ODBC Data Source with the name `default` and associating this with a database selected by the user. Refer to the Windows help provided with the Mimer Administrator for details on how to create User Specific Mimer ODBC Data Sources.

When a User Specific Mimer ODBC Data Source exists with the same name as a System Wide Mimer ODBC Data Source, the user-specific one takes precedence.

3.8.2 Troubleshooting Remote Database Connect Failures

If an attempt to connect to a remote database fails, the client/server connection can be tested by starting Mimer BSQL on the client node and attempting to connect to the database on the server node.

In the event of a connect failure, the following should be checked:

- If the connect was attempting to access the default database, check that a user-specific or node-specific default database is correctly defined on the client node, see *The Default Database* on page *37* for details on how this is done.
- Check that the database been correctly set up as a local database on the server node, see *The Local Database* on page *28*, and as a remote database on the client node, see *Accessing a Database Remotely* on page *29*, and that the name of the remote database is the same as that of the local database.

Linux: Verify that the `inetd` daemon is listening to the mimer TCP/IP service by using the `netstat -a` command.

- Check that the operating system user who is trying to establish the connection can access all required files etc. on the client node.

Linux & VMS: Check that the operating system user has read access to the `SQLHOSTS` file on the client machine.

- Check that the operating system user who is trying to establish the connection has all the required operating system privileges

VMS: Check that an operating system user who is trying to use DECNET has `TMPMBX` and `NETMBX` privileges enabled.

- If the TCP/IP protocol is being used, check that the server node is reachable from the client node over the network by using the `ping` command:

  ```
  ping server_node
  ```

- If the TCP/IP protocol is being used, try to telnet to the TCP/IP port. You should get a connection and when <CR> is entered, the connection should be closed by the server:

  ```
  telnet server_node 1360
  ```

3.8.2.1 NAMED PIPES

Win: If using `NamedPipes`, the operating system user must have an account set up on both the local machine and on the machine where the remote database resides. Both accounts must be set up with the same password.

If using `NamedPipes` to connect a Mimer SQL version 7.3 client to a Mimer SQL database server version 8 or later, it will be necessary to take certain steps to enable network communication.

Under version 7.3 the expected Service name was `MIMER`, but since version 8 the expected Service name is the name of the database.

Therefore, one of the following must be performed before a version 7.3 client can communicate with a newer remote database server:

1) On each version 7.3 client node, the Service parameter in the remote database definition must be changed to be the name of the database instead of the name `MIMER`.

Or

2) On the server node, start a `NamedPipes` server which listens to service `MIMER` so that it can redirect communications to the correctly named database server.

If using `NamedPipes` to connect a Mimer SQL client version 8 or later to a Mimer SQL version 7.3 database server, the Service parameter in the remote database definition on the client node must be changed to the name `MIMER` instead of the name of the database.

3.9 Executing SQL Statements

To execute SQL statements you can use DbVisualizer or BSQL, both included in the Mimer distribution.

The Mimer BSQL program supports a number of command-line arguments. See *Mimer SQL System Management Handbook, Chapter 9, Mimer BSQL* for a detailed description.

The syntax for Mimer BSQL (expressed in Unix-style) is as follows:

```
bsql [-u[username]] [-p[password]] [-s|-m] [-qquery] [database]
```

If neither `-s` nor `-m` is specified for the optional mode flag, the way the database is accessed will be determined by the setting of the `MIMER_MODE` variable, see *Specifying Single-user Mode Access* on page *151*, or, if this is not set, it will be accessed in multi-user mode.

Arguments containing more than one word should be enclosed in ". Note that VMS translates arguments to lower case when Unix-style syntax is used.

Linux:	The Unix-style command-line flags must be used on a Linux or macOS machine.

VMS:	Either the Unix-style or the VMS-style command-line flags may be used on an OpenVMS machine – see the *Mimer SQL VMS Guide* for more details.

Win:	The Unix-style command-line flags can be used from a command prompt window.

Chapter 4

Managing a Database Server

The database server enables one or more users to access a database. Each database may have one database server running against it and the server runs on the machine where the database resides.

The parameters which control a database server and which can be tuned to optimize performance are specified as part of the definition of a local database.

The `MIMCONTROL` functionality provides facilities for managing the operation of a database server (e.g. starting, controlled shutdown, etc.).

The `MIMINFO` functionality provides facilities for getting system management information from a Mimer SQL database server, such as:

- listing details for the users on the system
- monitoring performance parameters
- dumping data for trouble-shooting analysis by Mimer support personnel.
- listing SQLPOOL parameters

Operational and error messages generated by a database server are recorded in the database server log, see *Database Server Log* on page *65*.

Mimer SQL Database Servers

A database, with its set of databank files, can be operated by different types of database servers. The database server most commonly used is the standard Mimer SQL Experience database server called mimexper. The following are some server types that may be available, depending on the Mimer SQL product installed:

- mimexper - Standard Mimer SQL database server program.
- miminm - In-memory Mimer SQL database server program.

To select database server for a specific database, the `ServerType` parameter found in the database configuration option is used (see *The Local Database* on page *28*).

Mimer SQL Experience Database Server

The Mimer SQL Experience server is the standard version for the new generation of Mimer SQL database servers with many innovative and fundamental improvements. See Release Notes provided with each distribution and the Technical Description for Mimer SQL version 11 (or later), found on the Mimer SQL developer site, for the details.

Mimer SQL In-memory Database Server

For systems where a relational database and extreme performance is looked for, the Mimer SQL In-memory database server is the choice. This database server works in memory only, thus providing a huge throughput.

To save a database state to continue from at a later stage an online backup (see *Online Backup Commands* on page *72*) can be executed. An online backup will, while the system is running, produce a complete and consistent database file setup written to disk - a set of files that the in-memory database server can use to start from at next start-up.

Please note that because all data is stored and managed exclusively in main memory, **all data will be lost when the server is stopped, and upon a process or server failure**. Thus it is recommended having a procedure of doing Mimer SQL Online Backups if start-up data should be maintained.

System Performance

In a Mimer SQL system there are a number of system-wide parameters that have a major impact on the overall system performance.

The most important parameters are the bufferpool size and the number of request and background threads.

Database Server Memory Areas

The database server memory requirements include the following components:

- Code
- Data and thread stacks
- Bufferpool
- Communication buffers
- SQLPOOL

Code

The server code requires usually a few Mb, but it depends on the platform.

Data and Thread Stacks

As a rough guideline, assume about 500 Kb data plus 400 Kb for each thread started (the total number of threads started is the number of background threads plus the number of request threads), the actual figures, however, depend on the operating system being used.

Bufferpool

The bufferpool is the main primary memory cache used by the basic data access routines in the Mimer SQL database management and contains data pages from the databank files. It is a local memory area in the database server process.

The bufferpool does not grow dynamically, so whenever the bufferpool is full and access to a new page is required, space is released in the bufferpool by swapping out the least-recently-used resident page.

Frequent page replacement operations detract from the overall system performance since access to disk is relatively slow. The best Mimer SQL performance is thus obtained by having as large a bufferpool as possible without exceeding the amount of main memory available. In practice, it is always necessary to find a suitable compromise between allocation of memory to the Mimer SQL bufferpool and keeping memory available for user applications and operating system tasks.

The size of the bufferpool depends on the parameters *Pages4K*, *Pages32K* and *Pages128K* which are specified as part of the local database definition, see *The Local Database* on page *28*.

The amount of memory used by the database buffers can be calculated by:

buffer space in kilobytes = *Pages4K**4 + *Pages32K**32 + *Pages128K**128

Note: The bufferpool contains a variety of other data, therefore the total bufferpool size will be at least 10% greater than the space needed for the database buffers.

The default initial bufferpool size for a database server is based on the memory available on the machine.

Fine tuning of the bufferpool is performed manually by adjusting the parameters in the local database definition, see *The Local Database* on page *28*, after the Mimer SQL system is fully installed and has been functional for a period of time. The fine tuning should be repeated whenever there is a significant change in the computer workload distribution.

Since the Mimer SQL bufferpool size affects the performance of both Mimer SQL and other applications (because it reserves memory for a Mimer SQL database server), it is advisable to perform regular routine checks on the bufferpool statistics in an operational system by generating a Performance report, see *The Performance Report* on page *59*.

Note: The Windows NT performance monitor can also be used to monitor a database server running on any platform. Refer to the documentation supplied by Microsoft for the Windows NT operating system for details.

Bufferpool Tuning Guidelines

Some general guidelines for bufferpool tuning are:

- Whenever main memory is available, it should be allocated, if possible, to the bufferpool.
- Ensure that the bufferpool is not subject to system paging or swapping, since the paging algorithms used by Mimer SQL and the operating system usually differ, and forced cooperation between the two will often detract considerably from Mimer SQL's performance.
- If more than about 2% of all Mimer SQL page requests result in a page fault, the bufferpool is too small. Statistics for page requests and faults are presented in the Performance report, see *MIMINFO - System Information* on page *57*.
- It is important to take note of the page fault statistics for each region in the bufferpool to ensure that the most appropriate allocation has been made in each.

  The Mimer SQL system decides which page size is most appropriate for each task to be performed. For example, 32K pages are currently used for transaction data (this may change in the future) and therefore allocating too few 32K pages may currently adversely affect performance even though generous allocations have been made in the other bufferpool regions.

Communication Buffers

Each communication buffer is about 70 Kb, it varies slightly depending on platform.

There is one communication buffer for each user as defined by the Users parameter in the local database definition, see *The Local Database* on page *28*.

All communication buffers reside in shared memory.

SQLPOOL

The SQLPOOL area contains information about opened tables and databanks, compiled SQL programs, etc.

The initial size (in Kb) of the SQLPOOL is determined by the *SQLPool* parameter in the local database definition, see *The Local Database* on page *28*.

The SQLPOOL area grows dynamically when the database server needs more space. The local database parameter `MaxSQLPool` controls the maximum size (in Kb) of the SQLPOOL.

The value for `MaxSQLPool` is 2000*(Users+RequestThreads) by default.

The SQLPOOL area is never locked in physical memory. This allows the SQLPOOL to grow dynamically and it may become larger than the physical memory allocated to the server process. The operating system generally manages this situation by page-faulting. The page-faults will not affect bufferpool performance if that area is locked in physical memory.

If the amount of operating system page-faulting observed in a database server becomes excessive, it is an indication that the memory required by the server process is much greater than the amount of physical memory allocated to it. In this case, either more memory must be installed on the machine or the local database parameters controlling memory allocation must be adjusted to reduce the memory required by the database server process.

Threads

The Mimer SQL database server process supports a number of separate request threads and background threads, running simultaneously under the operating system.

Number of Request Threads

The amount of concurrency that the database server can support is dependent on the number of available request threads. If there are more concurrent requests than threads, the database server will start scheduling requests to improve response times. Increasing the number of request threads in a situation like this may improve performance.

The number of request threads in a database server is defined at system start-up. A change to the number of request threads requires that the system be stopped and re-started.

The maximum number of concurrent request threads is limited by the size of the bufferpool.

Number of Background Threads

The background threads in a Mimer SQL database server perform tasks such as:

- Recording transactions in `LOGDB`.
- Updating master and shadow databanks.
- Securing data on disk.
- Online backup.

See *Background Threads* on page *62* for information relevant to fine-tuning the number of background threads.

Network Encryption

With network encryption enabled, Mimer SQL uses AES-GCM (Advanced Encryption Standard with Galois/Counter Mode) to encrypt network communication between the database server and its clients. Communication over TCP/IP is encrypted, while communication using other protocols is unaffected.

AES-GCM provides authenticated encryption (confidentiality and authentication) by encrypting the network communication packages and by creating a MAC tag over the encrypted data. The authenticity of the data is provided by the MAC tag, which ensures that the data has not been altered or tampered with during transmission.

Network encryption is enabled by setting a parameter named `NetworkEncryption` in the database server.

Linux + VMS:	`NetworkEncryption` is a `MULTIDEFS` parameter. For details, see *MULTIDEFS Parameters* on page *163*.

Win:	The `NetworkEncryption` parameter can be set in Mimer Administrator.

Network encryption is available from version 11. In the case of an older Mimer SQL database server being upgraded to version 11, a password change is needed for IDENTS created before the version 11 upgrade, to be able to use encrypted communication over TCP/IP.

The command `miminfo -V` can be used to see the network encryption status of connected clients.

Database Server System Requirements

From the point of view of the operating system, a database server requires the system resources described in the following sections.

Physical Memory

The amount of physical memory used by the database server process is determined by parameters in the local database definition, see *The Local Database* on page *28*, whose initial default values are determined by looking at the amount of installed memory.

VMS: For a database server running on an OpenVMS node the amount of physical memory used by the database server process will vary between the OpenVMS process parameters `WSQUOTA` and `WSEXTENT`.

The `WSQUOTA` parameter is calculated by `MIMCONTROL` and is set large enough to include the bufferpool, initial `SQLPOOL`, communication buffers, code, and stack data.

The `WSEXTENT` parameter is set to the `SYSGEN` parameter `WSMAX` (the maximum amount of physical memory a single process may have).

For large buffer pools, it is recommended that a resident memory reservation created in VMS. Please see the *Mimer SQL OpenVMS Guide*.

Virtual Memory

The amount of virtual memory that the database server process can use is limited by the operating system.

Linux: The virtual memory handling on Linux platforms is platform specific – refer to the documentation for the specific Linux operating system you are using. (Often a paging file used).

VMS: The `MIMCONTROL` command sets the paging file quota of the database server so that it is large enough to contain all memory areas, including the bufferpool and an SQLPOOL that has grown to `MaxSQLPool` kilobytes.

It may be appropriate to create larger page files to increase the amount of virtual memory available to the database server.

Note that if the buffer pool does not use the page files if it is placed in a resident memory reservation.

Win: If you get a message saying the system is running out of virtual memory you may need to increase the size of your paging file. This done by using the Virtual Memory option in the Performance section of the System control panel.

Global Pages

VMS: The database server creates a global section for its communication buffers. This global section resides on the page file. The amount of memory a global section may take from a page file is generally controlled by an operating system parameter. If this limit set by the operating system is exceeded, the MIMCONTROL/START command will fail with the message:

```
%SYSTEM-E-EXGBLPAGFIL, exceeded global page file limit
```

If this happens, the OpenVMS SYSGEN parameter called GBLPAGFIL, which limits the amount of memory that global sections may take from the page files, should be increased.

MIMCONTROL - Controlling the Database Server

MIMCONTROL functionality is supplied on all platforms as a complete administration tool for managing database servers.

Linux: The database servers on a Linux node can be controlled using the mimadmin command (see the mimadmin man-page). This command invokes the MIMCONTROL program, and other programs, as required. The MIMCONTROL command can also be used directly under Linux.

A database server on Linux can be administered by the owner of it or by the superuser root. To change ownership of a database the mimdbfiles command is used (see the mimdbfiles man-page).

When a database server on a Linux machine is started for the first time, MIMCONTROL will create a default multidefs file containing appropriate default parameter values, based on the amount of memory installed on the machine. Refer to *The MULTIDEFS Parameter File* on page *161* for details.

Database server performance can be fine-tuned later by adjusting the parameters as required.

VMS: The database servers for the local databases on an OpenVMS node are controlled by using the MIMCONTROL command directly (as described in this section).

When a database server on an OpenVMS machine is started for the first time, MIMCONTROL will create a default MULTIDEFS file containing appropriate default parameter values, based on the amount of memory installed on the machine. Refer to *The MULTIDEFS Parameter File* on page *161* for details.

Database server performance can be fine-tuned later by adjusting the parameters as required.

Win:	The Mimer Administrator can be used to control database servers on Windows platforms. Database servers are also controlled by using the Mimer Controller utility. Refer to the Windows help provided with the Mimer Controller for further details. You must belong to the administrators group to control database servers. It is also possible to use the Windows commands `NET START`, `NET STOP`, etc. to control database server processes.

Syntax

`MIMCONTROL` is controlled by flagged information specified on the command-line.

The overall syntax for `MIMCONTROL` (expressed in short form Unix-style) is:

```
mimcontrol [-bcdegkAL] [-l chan] [-s [secs]] [-t [secs]] [-w [secs]] [-r type]
    [database]

mimcontrol [--dcl] [--status] [--disable|--enable] [--generate] [--kill]
    [--mcs] [--logout chan] [--start[=secs]] [--stop[=secs]] [--wait[=secs]]
    [--dump] [--report=type] [database]

mimcontrol [-v|--version] | [-?|--help]
```

If `MIMCONTROL` is invoked without any options, it displays help options on the command-line.

Command-line Arguments

Unix-style	VMS-style	Function
`-b` `--dcl`	`/STATUS/DCL`	Output status information about the specified database server as a single-line list suitable for use in a script. For details about the output string resulting from this option, see *Database Server Status* on page *54*.
`-c` `--status`	`/STATUS`	Output status information about the specified database server. This option can be combined with the `-s` option, see *Examples* on page *55*.
`-d` `--disable`	`/DISABLE`	Disable new user connections to the database server. Users already connected are not affected. This option can be combined with the `-s`, `-t` and `-w` options, see *Examples* on page *55*.
`-e` `--enable`	`/ENABLE`	Enable new user connections to the database server.
`-?` `--help`	`/HELP`	Show help text.
`-k` `--kill`	`/KILL`	Kill the database server immediately. This should only be used in emergency situations when a normal stop using the `-t` option does not work. The next time the database is started, all databanks that were open at the time the server was killed will be automatically checked. Connected users will receive an error the next time they attempt to access the database.
`-l chan` `--logout=chan`	`/LOGOUT=chan`	Force logout of the specified channel number. Use channel numbers displayed by the `USERS` option of the `MIMINFO` command, see *The Users List* on page *59*.
`-g` `--generate`	`/GENERATE`	Generate a default multidefs file. **Win:** This switch is not available in the Windows environment.

Unix-style	VMS-style	Function
`-s [timeout]` `--start[=timeout]`	`/START[=`*timeout*`]`	Start the database server. If the server does not become operational within the specified number of seconds, the server will be killed. Default timeout is 600 seconds. This option can be combined with the `-c` and `-d` options, see *Examples* on page *55*.
`-t [timeout]` `--stop[=timeout]`	`/STOP[=`*timeout*`]`	Stop a database server. Any remaining users will be logged out. If the server does not stop within the specified number of seconds, the server will be killed. The default timeout is 120 seconds. This option can be combined with the `-d` and `-w` options, see *Examples* on page *55*.
`-w [timeout]` `--wait[=timeout]`	`/WAIT[=`*timeout*`]`	Wait for all connected users to log out. If there are still users connected after the timeout period expires, the command fails. The timeout period should be given in seconds. If no timeout period is specified wait will be performed without any timeout. This option can be combined with the `-d` and `-t` options, see *Examples* on page *55*.
`-A` `--dump`	`/DUMP`	Create a dump directory and produce dumps of all internal database server areas to files in that directory. The files produced can be examined by using the `MIMINFO -f` command, see *MIMINFO - System Information* on page *57*.

Unix-style	VMS-style	Function
`-h` `--hold`	`/HOLD`	**VMS:** The command `MIMCONTROL/START` starts a database server. Adding the `/HOLD` qualifier will cause the `MIMCONTROL` command to wait for the started server to stop before returning control to the DCL prompt. This simplifies writing scripts for automatic server restart. When the `/HOLD` qualifier is used, the `MIMCONTROL` command will exit with the final status code of the stopped database server process.
[*database*]	[*database*]	Specifies the name of the database to access. If a database name is not specified, the default database will be controlled. The default database is determined by the setting of the `MIMER_DATABASE` environment variable. The `DEFAULT` setting in `SQLHOSTS` is not used for `MIMCONTROL`.

Linux: The Unix-style command-line flags must be used on a Linux machine. Both short form switches (e.g. `-s`), and long form switches (e.g. `--start`) are supported.

VMS: Either the Unix-style (short or long form) or the VMS-style command-line flags may be used on an OpenVMS machine – see the *Mimer SQL VMS Guide* for more details.

Win: The Unix-style command-line flags can be used from a Command Prompt window. Both short form switches (e.g. `-s`), and long form switches (e.g. `--start`) are supported.

Database Server Status

The `mimcontrol -b` (`MIMCONTROL/STATUS/DCL`) command is a special form of the `mimcontrol -c` (`MIMCONTROL/STATUS`) command which returns the database server status information in the form of a single string containing a comma-separated list which is useful when writing scripts.

Linux: On Linux, the string returned from `mimcontrol -b` can be piped and processed as required. The Linux command `cut` can be used to extract the list elements, e.g. the following command which will print the second list element:

```
# mimcontrol -b | cut -f2 -d ','
```

VMS: On OpenVMS, the `MIMCONTROL` command is silent and sets the DCL symbol `MIMER_STATUS` to the value of the status string.

The lexical function `F$ELEMENT()` can be used to extract the list elements, e.g. the following command will extract the second list element:

```
$LOGINS=F$ELEMENT(1,",",MIMER_STATUS)
```

Win: On Windows, the string returned from `mimcontrol -b` can be piped and processed as required, depending on the script language being used.

Status String Components

The status string has the following components:

```
server-state,logins,db-directory,connections,server-pid,start-date,bpool-size
```

Each component is described below:

Component	Description
`server-state`	The server-state value shows the state of the database server. The value is one of: stopped, starting, running, stopping or crashing.
`logins`	The logins value is either: enabled, if new logins are permitted, or disabled, if the server has been ordered to reject new logins.
`db-directory`	The db-directory field shows the home directory for the database.
`connections`	The connections field shows the number of clients connected to the server.
`server-pid`	The server-pid field gives the process id of the database server process.
`start-date`	The start-date field gives the date and time when the database server was started.
`bpool-size`	The size of the buffer pool (in bytes).

Note: If the database server is not operational, the status string will contain empty fields.

Examples

The parameter options can be combined in the following ways, examples are given in both VMS-style and Unix-style:

- Start a database server, but disallow new logins immediately:

```
MIMCONTROL/START/DISABLE db_server_name
mimcontrol -sd db_server_name
mimcontrol --start --disable db_server_name
```

- Start a database server and output a status message for the newly started server:

```
MIMCONTROL/START/STATUS db_server_name
mimcontrol -sc db_server_name
mimcontrol -start --status db_server_name
```

- Disable new user connections, then wait for up to three minutes for all users to log out.:

```
MIMCONTROL/DISABLE/WAIT=180 db_server_name
mimcontrol -dw 180 db_server_name
mimcontrol --disable --wait=180 db_server_name
```

  The exit code from the `MIMCONTROL` command is success if all users logged out within the three minute timeout period.

  If the timeout period expires and there are still users logged in on the system, the `MIMCONTROL` command will exit with a warning status code.

- The following command will wait for all users to log out of the system:

```
MIMCONTROL/WAIT/STOP db_server_name
mimcontrol -wt db_server_name
mimcontrol --wait --stop db_server_name
```

  When all users are logged out, the system will be stopped. If the wait timeout period expires, the `MIMCONTROL` command will exit with a warning status code without stopping the system.

- The following command is similar to the previous one, but will ensure that no new users log in to the system while waiting for all users to log out:

```
MIMCONTROL/DISABLE/WAIT/STOP db_server_name
mimcontrol -dwt db_server_name
mimcontrol --disable --wait --stop db_server_name
```

Exit Codes

`MIMCONTROL` returns a status code to the environment executing the command. The status code can be examined by scripts.

> **VMS:** On OpenVMS, the status codes correspond to the OpenVMS condition code severity levels.
>
> Use the `$SEVERITY` symbol in DCL command procedures.

The following return codes are used:

Linux/Windows	VMS	Usage
`0 (success)`	`1 (success)`	This code is used when the `MIMCONTROL` command has executed all options with no problems.
`1 (warning)`	`0 (warning)`	The warning code is used when there was a timeout in one of the options. The complete sequence of options may not have been executed.

Linux/Windows	VMS	Usage
> 1 (error)	2 (error)	The error code is used when the specified command could not be executed at all. For instance, if there was an illegal combination of options, or if the specified database name was not found. If an error status code is returned, an informational error message will also be produced.

MIMINFO - System Information

MIMINFO is used to obtain information from a Mimer SQL database server which is useful for system control, system tuning and trouble-shooting analysis.

Information can be generated from an active Mimer SQL database server as well as from the SQLPOOL and bufferpool dump files produced by using MIMCONTROL, see *MIMCONTROL - Controlling the Database Server* on page *50*.

The output from MIMINFO can be displayed on the screen and may also be directed to a file.

The following reports may be obtained from MIMINFO (further details on each report can be found in the sub-sections that follow):

Report	Description
Users list	this lists details of all the users currently connected.
Performance report	this provides information useful for monitoring performance parameters (MIMSERV).
Bufferpool report	this produces a report which is useful to Mimer SQL support personnel when investigating system problems (MIMDUMP).
SQLPOOL report	displays SQLPOOL parameters
Version report	displays version related information for a started server and its client connections

Syntax

The MIMINFO program is controlled by flagged information specified on the command-line.

The overall syntax for MIMINFO (expressed in short form Unix-style) is:

```
miminfo [-o file] [-m [-b bcbs]] | -p | -s | -u [-f | database]

miminfo [--output=file] [--mimdump [--bcblimit=bcbs]] | --performance |
    --sqlpool | --users [--file | database]

miminfo [-v|--version] | [-?|--help
```

Command-line Arguments

Unix-style	VMS-style	Function
`-b bcbs` `--bcblimit=bcbs`	`/BCBLIMIT=bcbs`	Limits the displayed bcb list. Used together with the `--mimdump` option.
`-f` `--file`	`/FILE`	Take information from a dump file (for a users list, a dump file called `sqlpool.mdmp` is expected to exist otherwise a dump file called `bpool.mdmp` is expected to exist)
`-m` `--mimdump`	`/MIMDUMP`	Produce Bufferpool report (`MIMDUMP`)
`-o file` `--output=file`	`/OUTPUT=file`	Send output to the specified file instead of to the screen
`-p` `--performance`	`/PERFORMANCE`	Produce Performance report (`MIMSERV`)
`-s` `--sqlpool`	`/SQLPOOL`	Display SQLPOOL parameters
`-u` `--users`	`/USERS`	Display users list
`-V` `--version`	`/VERSION`	List version information.
`-?` `--help`	`/HELP`	Show help text.
`database`	`database`	Take information from the specified database. If a database name is not specified, the default database will be accessed, see *The Default Database* on page *37*.

Linux: The Unix-style command-line flags must be used on a Linux machine. Both short form arguments (e.g. `-u`), and long form arguments (e.g. `--users`) are supported.

VMS: Either the Unix-style or the VMS-style command-line flags may be used on an OpenVMS machine – see the *Mimer SQL VMS Guide* for more details.

Win: The Unix-style command-line flags can be used if the miminfo program is run from a Command Prompt window. Both short form arguments (e.g. `-u`), and long form arguments (e.g. `--users`) are supported.

The shortcut `Mimer Info` can also be used to run the program and interactive selections can then be made in the program.

A detailed description of each of the `MIMINFO` reports follows.

The Users List

A users list can be generated from an active database or from a dump file produced using `MIMCONTROL`.

```
miminfo [--output=file]  --users  [database] | --file
```

The users list shows the name of each ident connected to the database, the channel number used by the connection, the state of the connection, transaction number, the name of the operating user, the network communication protocol (or 'local') and node identification information for connected machine.

The channel number may be used in conjunction with `MIMCONTROL` to kill a user.

The following is an example of a users list report:

```
Username        Channel State  Trans. no OS user      Prot   From
============ ========== ===== ========== ============ ====== ====
SYSADM            16387 Busy           3              TCP    204.71.200.67
SYSADM            16388 Busy             STELLA       Local  00019120

Total of 2 users
```

The Performance Report

The performance report can be used by the system administrator to monitor performance parameters during Mimer SQL use. The Performance report can be generated from an active database or from a dump file produced using `MIMCONTROL`.

```
miminfo [--output=file]  --performance  [database] | --file
```

The performance report presents five kinds of statistical information which may be useful for system tuning (statistics for page management, transactions, background threads, databank and table usage).

Note: When a performance report is used as an aid to system tuning, it is important that the report is generated when the database is in full use. The output from several executions over a period of a few hours or days can provide valuable information on fluctuations in system usage.

The performance report contains the following information:

- General Statistics
- Page Management Statistics
- Transaction Management Statistics
- Background Threads
- Databank Statistics
- Table Statistics

General Statistics

The following table lists the general statistics information available. Where applicable, we have provided a detailed description

Statistics	Description
Current date and time	When the statistics was generated.
Current MIMER/DB version	Mimer server version.
Starting date and time	When the Mimer server was started.
Current hardware and operating system	Computer hardware and OS information.
System status	If the database server is in an error state, a database dump is usually made automatically. It can be made manually by using the `-A` option with `MIMCONTROL`. The dump directory created should be saved for use by Mimer SQL support personnel. The database server can then be restarted. The database server log, see *Database Server Log* on page *65*, should also be inspected to help find the cause of the failure.
Error count	Number of errors that have been written to the database server log. This value should normally be zero.
No. of request threads	Number of request threads started, see *Number of Request Threads* on page *47*.
No. of background threads	Number of background threads started, see *Number of Background Threads* on page *47*.
No. of I/O threads	Number of I/O threads (typically zero on most machines where separate threads are not needed for I/O processing).

Page Management Statistics

Statistics	Description
No. of pages written to disk	An indication of the frequency of disk update operations.
No. of file extend operations	The total number of times databank files have been dynamically extended since the latest startup. The value should preferably be as low as possible for performance reasons. It is possible to check databank size usage with the `DESCRIBE` command in Mimer BSQL. A databank can be extended by using the commands: `ALTER DATABANK ADD …` or `ALTER SHADOW ADD ….`

Statistics	Description
Buffer size 4K (32K, 128K)	The bufferpool is divided into a region with 4K buffers, one region with 32K buffers, and one region with 128K buffers.

The following information is given for each region:

Statistics	Description
No. of page buffers	This is the number of page buffers allocated to this bufferpool region.
No. of page buffers per sorter	Total number of Mimer SQL pages that a request thread performing a sort operation may utilize.
No. of remaining sorters	The initial value specifies the number of concurrent sort/merge steps that are allowed.
No. of page partitions	Each region in the bufferpool is divided into separate partitions. Each partition can be accessed concurrently by the Mimer SQL request threads. In tightly coupled multi-processor systems it is desirable, for performance reasons, to have at least as many partitions as there are CPUs. The number of partitions may be increased by increasing the region size.
No. of page requests	Total number of access operations to pages in the buffer region since the latest system start-up.
No. of page faults	Total number of page access requests that resulted in disk read operations. If this value is more than about 2% of the total number of page requests, performance may be improved significantly by increasing the bufferpool size.
No. of pages swapped out	Total number of pages which were written to disk when they were swapped out of the buffer region.

Transaction Management Statistics

Statistics	Description
No. of transaction commits	Total number of successful read/write transaction commits since the latest system start-up.
No. of read commits	Total number of successful read-only transaction commits since the latest system start-up.
No. of transaction checks	A high proportion of transaction checks in relation to the total number of transactions may indicate ill-designed application programs, with long transactions that are more likely to give rise to transaction conflicts.
No. of transaction aborts	Total number of transactions aborted by the optimistic concurrency protocol since the latest system start-up. User requested transaction aborts are not counted.

Statistics	Description
No. of pending restarts	This is an indication of how much information is stored in `TRANSDB`. Number of restarts is counted for each databank used in a transaction. This figure grows larger when shadows are set offline. If all databanks have been accessed and there are no offline shadows there should not be any pending restarts.

Background Threads

Statistics	Description
SWA	Background thread identifier.
State	State of the background thread. If the background thread is currently working with a transaction, `active` is displayed. If the background thread is not doing anything, `inactive` is displayed. `I/O processing` means that the background thread is flushing one or more transactions to disk. `unused` means that the background thread is allocated but not currently running (i.e. the thread is not started or closed down).
Trans-no	The number of the transaction currently being processed.
Trans-count	The number of transactions processed by the background thread.
Pending background thread requests	This indicates how many transactions have not yet been processed by the background threads. If there are too few background threads this value will grow.
Application waiting for trans-no	For certain operations (`SET DATABANK OFFLINE`, for example) the application has to wait for the background threads to complete their operations. If there are too few background threads, it may take some time before this operation is complete. By comparing this `trans-no` with the `trans-no` being handled by the background threads it is possible to see how many transactions are left before the operation is completed.

Databank Statistics

Statistics	Description
Name	The name of the databank or shadow.

Statistics	Description
DBANKID, SEQNO	Databank identification. These two values correspond to the columns `DATABANK_SYSID` and `DATABANK_SEQNO` in the data dictionary table `SYSTEM.DATABANKS`.
Type	The databank option `WORK`, `TRANSACTION`, `LOG`, or `READ ONLY`. See *Re-creating TRANSDB, LOGDB and SQLDB* on page *77*. The `SQLDB` databank has the type `TEMPORARY`, and shadows have the type `SHADOW`.
Users	Internal user count.
Access	Access mode by which the databank was opened. The possible values are: `Read`, `Write`, `Shared` and `Exclusive`. If the databank is open but not referenced by any active statement, `None` is displayed.
DB-Check	The `DB-Check` field indicates the progress of a databank check. The possible values are `Init`, `Working` (foreground processing, typically index check), `Wait B.` (foreground ready, waiting for background entrance), `Backgr.` (background processing), `Aborting`, `Aborted` or `Complete`. After the `DB-Check` field, a field for additional information may show up. The values here can be the shadow state: offline, or the online backup states: backup in progress or backup completion.
No. of databanks currently open	A count of both databanks and shadows opened in the system.
Max number of databanks open concurrently	This is defined by a parameter in the local database definition, see *The Local Database* on page *28*, or possibly by a limit in the operating system.
Databank verification count	Databank verification is automatically performed when a databank is re-opened without having been correctly closed. Each time this happens a log entry is written to the database server log. When a databank is verified the databank verification count is incremented. The count is cleared when the system is started, and a databank is only verified once per session.
Running background verifications…	Indicates the number of active databank verifications.
Pending background verifications…	Indicates the number of active databank verifications.

Statistics	Description
Databank verification is only done on index pages…	Indicates the databank verification mode as defined in the local database definition, see *The Local Database* on page *28*
Databank verification is performed on all pages…	Indicates the databank verification mode as defined in the local database definition, see *The Local Database* on page *28*

Table Statistics

Statistics	Description
No. of tables currently open	This shows the number of tables open in both master and shadow databanks. Also included are the read and write sets used by each user.
Max number of tables open concurrently	This number is set as a parameter in the local database definition, *The Local Database* on page *28*.

Bufferpool Report

The Bufferpool report is used by Mimer SQL support personnel for trouble-shooting when database problems are reported by customers.

```
miminfo [--output=file]  --mimdump  --file
```

SQLPOOL Report

SQL pool memory allocated is the amount of memory allocated from the operating system for the SQLPOOL. A part of that memory is in use by the server and is displayed on the row SQL pool memory used.

```
miminfo [--output=file]  --sqlpool  [database] | --file
```

The following is an example of an SQLPOOL report:

```
SQLPOOL report
==============
SQL pool memory allocated (KB):        1656
SQL pool memory used      (KB):         554
```

Version Report

A version report can be generated from an active database or from a dump file produced using MIMCONTROL. The report is generated using the following command:

```
miminfo [--output=file]  --version  [database] | --file
```

The version report displays information about the server and about each connected client.

The server information includes server type, version and platform.

The client information includes channel number, database API, version, network encryption and node identification information for the connected machine.

The channel number can be used to identify the connection when it appears in other reports.

The following is an example of a version report:

```
miminfo --version
M I M E R / M I M I N F O
Version 11.0.4A Sep 5 2020

Server type:      Mimer SQL Experience
Server version:   11.0.4A
Server platform:  Windows x64

   Channel Client interface Version  Platform     Encrypt  Prot    From
   ======= ================ =======  ========     =======  =====   ====
   1327109 ODBC             11.0.4A  Windows x64  none     Local   00001E2C
     32772 Embedded SQL     11.0.4A  VMS Itanium  AES/GCM  TCP     Freke

Total of 2 users
```

Database Server Log

The database server log lists startup and shutdown messages for the database server. It may also contain warning and error messages if such situations have been detected by the database server.

Linux + VMS: A log file called `mimer.log` is created when the database server is started for the first time. This file is located in the database home directory.

In addition, you can set the `Oper` parameter in the `MULTIDEFS` file to send e-mails containing serious database server messages to relevant people. These messages always go to the system log as well.

For more information, see *Appendix C The MULTIDEFS Parameter File*.

Win: Database server events are logged in the `EventLog` which may be examined using the Windows event viewer.

Several Installations on One Machine

Linux: Under Linux, a host computer may have several Mimer SQL installations, of the same and different versions, installed simultaneously.

If several Mimer SQL version 10 installations are available, only one of them can be linked to `/usr/lib` and `/usr/bin` at the same time.

To access an installation that is not linked to these locations, the environment variables `PATH` and `LD_LIBRARY_PATH` (or corresponding to located shared libraries) must be used explicitly.

For more information on environment variables, see *Mimer SQL - Getting Started on Linux*.

VMS: Under OpenVMS, a host computer may have several Mimer SQL installations, of the same and different versions, installed simultaneously.

Win: Only one Mimer SQL installation can exist on a computer running Windows.

Chapter 5

Backing-up and Restoring Data

This chapter discusses backup and restore of Mimer SQL databanks. Two types of backup procedures are described:

- System Backup, i.e. backing up the databank files from the host operating system. When using host operating system tools for doing databank file backup, the database server must be stopped in order to keep the database consistent.
- Online Backup, i.e. using the SQL system management statements. The main advantage of online backup is that the database server can continue to operate (backup operations are performed in the background).

Some of the discussion in this chapter refers to shadowing databases, see *Mimer SQL Shadowing* on page *123*, which is an optional Mimer SQL module that allows one or more copies of a databank to exist on different disks. Shadowing provides a high level of protection from disk failure because the system will automatically use a databank shadow if the master databank is lost, thus allowing normal database activity to continue without interruption. Databank shadows also allow a copy of a databank to be temporarily set offline (e.g. to be backed-up) without interrupting normal system use.

Several references to transaction handling are made in this chapter. If you are not familiar with transaction handling in Mimer SQL see the *Mimer SQL Programmer's Manual, Chapter 9, Transaction Handling and Database Security*.

Background Information

A Mimer SQL database consists of a collection of databanks (each in a separate operating system file) containing tables with data used by the applications. The `SYSDB` system databank contains a data dictionary describing the different objects in the database.

Note: Backup protection for `SYSDB` is particularly important for protecting the database, since `SYSDB` contains all information describing the database structure. If `SYSDB` is lost, the system must be rebuilt from scratch.

Database Consistency

Database consistency is handled on two levels: physical and logical.

Physical consistency means that the tables are readable by Mimer SQL. This is ensured as long as the databank file is not physically damaged.

Logical consistency means that the tables contain valid data. This is ensured by Mimer SQL's transaction handling. All transactions are saved in the TRANSDB databank during build-up and are applied to the databanks when they are committed. To use transaction handling, the databank must be created with the TRANSACTION or LOG option.

Transaction handling makes it possible to ensure that a user cannot commit a read write transaction which has read data that is being concurrently updated by another user. If a transaction is successfully committed then all operations in the transaction are performed. If the transaction is aborted due to a conflict, none of the operations in the transaction are performed.

The tables may be logically inconsistent if Mimer SQL is stopped before all operations in a committed transaction have been performed. At some time after the system is restarted, all uncompleted transactions will be read from TRANSDB for automatic completion. This happens in the background on a per-databank basis, after a databank is first accessed following the restart. Transactions that were not committed before the stop are aborted.

The DBOPEN facility, see *DBOPEN - Databank Open* on page *89*, can be used to open all databanks in one operation and thus achieve transaction consistency quickly.

LOGDB and TRANSDB Importance

Important: The information stored in TRANSDB is vital to keep the database consistent in all circumstances, not only in case of failure.

The LOGDB information will contain data on all the changes made to the databank from the time the backup copy of the databank was taken until the time of the disk crash. This information is used if a backup copy of a databank file is to be restored.

Note: Data changes that are not logged cannot be restored by this process, therefore it is important to consider the issue of transaction logging carefully.

If a databank becomes unavailable (because the Mimer SQL system is stopped deliberately or by a system failure) during the commitment of a transaction, information is retained in the TRANSDB system databank and used to complete the transaction when the databank becomes available again.

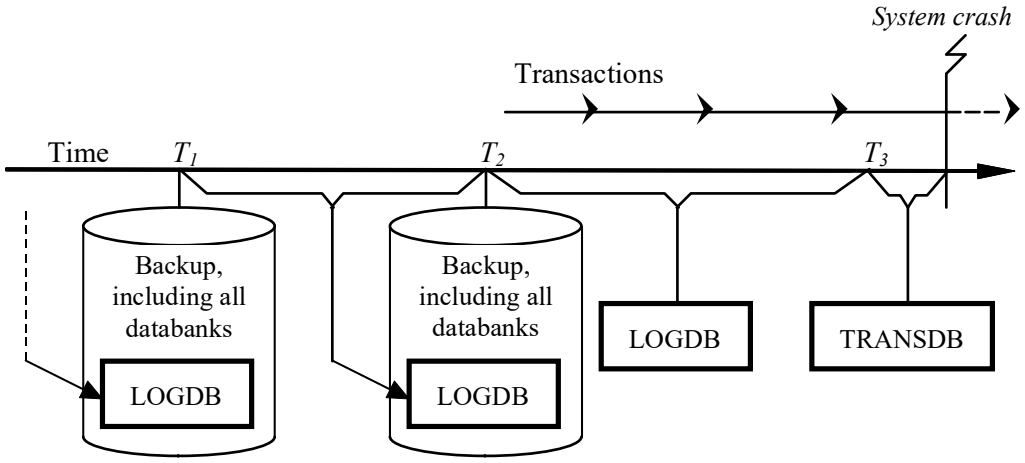
This is only true for databanks with the TRANSACTION or LOG option. Once information has been successfully written to both LOGDB and the databank file, it is removed from TRANSDB.

It is recommended practice to back up all the databanks of the database at the same time, and to ensure that LOGDB is always backed up whenever other databanks are backed up, because the LOGDB information provides the transaction data which links the previous backup copy of a databank with the databank as it exists at the current point in time.

Thus, when restoring a databank it should be brought to a state consistent with the latest backup. This is done either by using the latest backup copy of the databank or by using backed up LOGDB information with an older backup copy of the databank.

The current LOGDB system databank is then used to restore the final changes made between the time of the latest backup and the time the databank was lost.

Example



The graphic above describes a scenario which ends up in a system crash.

To recover from this situation the common operation is to start from the most recent backup (T_2) and then use the current LOGDB to recover data up to the state at T_3.

When the system is restarted, the current TRANSDB is used to automatically recover up to the moment of the crash.

If the most recent backup cannot be used, an older backup has to be brought in (T_1). This backup is restored up to the consistent state at T_2 by using the LOGDB stored in the backup at T_2.

Then the current LOGDB and TRANSDB are used to restore the transactions committed after the backup at T_2.

Note: Wherever possible, LOGDB should be stored on a different disk unit, with a separate disk controller, from the other databanks in order to minimize the risk that a disk crash or damaged disk controller destroys both the log and the other databanks.
LOGDB and TRANSDB should always be located on different physical disks which are ideally served by separate disk controllers and no other databank files should be located on either disk, since data may be lost if both TRANSDB and LOGDB are destroyed.
Refer to *Organizing Databank Files* on page *10*, for more details on data security and databank files.

Updates Recorded in LOGDB

The LOGDB system databank contains logged update information for each databank with LOG option.

It is recommended that all databanks, including LOGDB, are backed up at the same time and that LOGDB is cleared after backup by resetting the log. Thus, the backed up LOGDB will contain the information required to make databanks from the preceding backup consistent with the current backup.

This will provide double backup protection by allowing a lost databank to be recovered in one of the two ways listed below:

- restore the databank from the most recent backup and apply the updates currently held in LOGDB, or
- restore the databank from an earlier backup, then sequentially use the LOGDB files from each subsequent backup to make the databank consistent with the most recent backup, and finally apply the updates currently held in LOGDB.

The records in LOGDB should be cleared after a complete backup, in order to maintain consistency between the backup and LOGDB. This ensures that LOGDB only contains information about changes made to a databank since the last backup of it was taken. (It is possible to backup databanks without clearing LOGDB records, although care must be taken as this leaves the backup and LOGDB in an inconsistent state).

The ability to restore databank updates from a backup copy of LOGDB replaces the databank incremental backups which were supported in previous versions of Mimer SQL. These are still supported for backward compatibility but it is now recommended that LOGDB backups always be taken to offer the same double protection.

Caution: If, for any reason, the LOGDB databank is lost, no problems will be encountered immediately. All changes will have been properly recorded in the application databanks. A new, empty, LOGDB can simply replace the log that was lost.

However, a backup of the entire database must be taken immediately. The new LOGDB will be empty, and therefore in a state consistent with a backup of all databanks having just been taken and all LOGDB records cleared.

If a backup is not taken immediately, a later attempt to restore a databank is likely to fail because the restore operation will expect to find information in LOGDB that was lost when LOGDB was destroyed.

TRANSDB Considerations

TRANSDB requires backup protection since it nearly always contains unfinished transactions. If TRANSDB is lost before the Mimer SQL system is restarted, the database will be left in a logically inconsistent state.

Possible effects of losing TRANSDB before the database server is restarted are described in the following scenarios:

- If TRANSDB is lost, some of the databank updates that apply to the most recently committed transactions may have been made while others remain unfinished. The only safe operation to do to avoid a logically inconsistent database is to bring up the most recent backup copy and restore from LOGDB. In this case, the only loss is those transactions that were not completely written to LOGDB.
- If both TRANSDB and LOGDB are lost, the restoration, as described in the previous bullet, cannot be accomplished.

In the case where a restore is not possible, the best solution is to repair the inconsistency immediately after restarting the database server. This is done by using a tool such as BSQL for manual verification and update of data. This is usually possible if the user who initiated the interrupted transaction can be identified and contacted. (Many applications maintain a parallel audit log file for tracking purposes which can be used as a basis for repair work).

An alternative solution if both LOGDB and TRANSDB are lost, is to start over from the most recent backup of your databanks and reprocess all transactions since that time. This may be a costly operation.

Keeping TRANSDB and LOGDB on separate disks under separate disk controllers will minimize the risk that both databanks are lost at the same time.

A TRANSDB shadow is another possible security enhancement, see *Mimer SQL Shadowing* on page *123*.

Note: The TRANSDB system databank must never be deliberately deleted, because uncompleted transactions nearly always remain saved in the databank even if the database server is currently stopped.
If a TRANSDB file containing uncompleted transactions is deleted, inconsistency will occur because the information required to complete those transactions when the database server is re-started will have been lost.

SQLDB Considerations

The contents of SQLDB is transient, so this databank does not need backup protection.

However, it may be convenient to have SQLDB included in the backup so that a complete system can be restored easily, without any additional operations to recreate an empty SQLDB.

Some data retrieval requests in Mimer SQL may require large work areas or transaction handling areas for intermediate processing of the data (for instance, requests to sort or group large result sets will require large work tables in SQLDB). This is particularly relevant when ad-hoc queries may be submitted with little thought for the processing requirements or performance of the query. In systems where files expand automatically, the file for SQLDB can become very large as the result of one badly-planned query.

The databank attributes GOALSIZE and MAXSIZE are to manage databank file sizes. See *Mimer SQL Reference Manual, Chapter 12, CREATE DATABANK*.

Databank Backups

A databank backup is a copy of the databank file.

A databank backup is the starting point for any restore operation, and should be stored in a safe place separate from the working databank files (copied to a different disk or preferably written to backup media and removed from the machine).

The backup can be taken either by using the Mimer SQL system administration statements for online backup, see *Online Backups Using the SQL Statements* on page *74*, or by using the host file system utilities in a system backup, see *System Backups Using the Host File System* on page *75*.

After a backup is taken, the updates logged for the databank in question should be cleared from LOGDB. This will be done automatically when the SQL system management statements for online backup are used.

The DBC program, see *Databank Check Functionality* on page *81*, should be executed for each databank in the backup operation in order to validate the physical consistency of the databank.

For a system backup, the backup copies of the databanks should always be taken when the system is closed and the databanks are in a logically consistent state. That is, no uncompleted transactions should exist and all databanks should be backed up at the same time to safeguard database consistency.

System vs. Online Backups

The main advantage of online backup is that all databanks, including the system databanks, can be backed up while the system can remain operational. The backup is initiated and executed by use of SQL statements only. The disadvantage can be that there must be enough disk space available to copy the complete database.

If the disk space is limited, a system backup can be preferable. For a system backup, the database server must be stopped. A system backup needs certain SQL statements (such as set online/offline) to be used together with operating system commands for file copying, etc.

SQL Statements for Backing-up Databanks

Refer to the *Mimer SQL Reference Manual, Chapter 12, SQL Statements* for a detailed description and syntax definition of the SQL system management functions. A brief description of the purpose of each function appears here.

Online Backup Commands

The SQL system management statements that can be used to take backups are:

Command	Description
`START BACKUP`	starts a backup transaction.
`CREATE BACKUP`	creates a backup within a backup transaction. By default an online backup is created, but optionally an exclusive backup can be initiated, which will lock the databank from other users.
`COMMIT BACKUP`	commits a started backup transaction.
`ROLLBACK BACKUP`	aborts a backup transaction and ensures that all log records are preserved.

To use these statements to take a databank backup, the user must either be the creator of the databank, or have `BACKUP` privilege.

When the SQL statements are used to take a backup of a databank, the entire process of taking a databank backup is handled automatically.

The use of a backup transaction ensures that the backups taken within the transaction are consistent with one another, as each backup is effectively taken at the same point in time. Log records are cleared for successfully backed up databanks when the backup transaction is committed. If `LOGDB` is included in the backup transaction all log records are cleared.

Online/Offline Commands

The SQL system management statements (typically used when taking databank backups using the host file system) that can be used to set a databank, shadow or the whole database online or offline are:

Command	Description
`SET DATABASE OFFLINE`	sets all non-system databanks offline, and makes the database unavailable. If one of the databanks cannot be set offline (e.g. because it is being used), the command will fail.
`SET DATABASE ONLINE`	sets all databanks online, optionally clearing all records from `LOGDB` and makes the database available.
`SET DATABANK OFFLINE`	sets a databank offline and the databank pages are updated with all changes made by committed transactions so far. The databank file is closed (except `SYSDB`, which always remains open as long as the database server is running) so the file can be copied, and it becomes unavailable to database users.
`SET DATABANK ONLINE`	sets a databank online, making it available, optionally clearing records from `LOGDB`.
`SET SHADOW OFFLINE`	sets a list of shadows offline, making them unavailable.
`SET SHADOW ONLINE`	sets a list of shadows online, making them available, optionally clearing records from `LOGDB`.

A user setting the database online/offline, must have `BACKUP` privilege and must be the only user accessing the database.

A user setting a databank or a shadow online/offline, must either be the creator of the databank or have `BACKUP` privilege.

Restore Command

The SQL system management statement used to recover a databank in the event of it being damaged or destroyed is:

Command	Description
`ALTER DATABANK RESTORE`	used to restore a databank from a backup copy by using a `LOGDB` backup and/or the information currently in the `LOGDB` system databank.

A user using this function to restore a databank must be the creator of the databank or have `BACKUP` privilege.

Backing-up Databanks

This section describes procedures for taking databank backups and for restoring a databank in the event of it being damaged or destroyed.

Online Backups Using the SQL Statements

The procedure for taking databank backups using the SQL system management statements is detailed below.

A CREATE BACKUP statement is executed for each databank to be backed up and databank consistency is ensured by starting a backup transaction using the START BACKUP statement.

The backup transaction is committed by using the COMMIT BACKUP statement, which will perform the backup and clear the relevant LOGDB records. The ROLLBACK BACKUP statement can be executed to abort the backup transaction, which will preserve LOGDB.

Note: The databank check functionality (the DBC program) should be run before archiving the backup copies of the databank files (e.g. copying them to CD/RW) to verify the physical integrity of the databank files.

To backup databanks online, do the following:

1 Perform SQL statements for initiating and executing the backup.

```
SQL> START BACKUP;
SQL> CREATE BACKUP IN 'backup-file-name' FOR DATABANK databank-name;
 .
 .   (repeat for each databank to be backed up)
 .
SQL> CREATE BACKUP IN 'backup-file-name' FOR DATABANK logdb;
SQL> CREATE BACKUP IN 'backup-file-name' FOR DATABANK sysdb;
SQL> CREATE BACKUP IN 'backup-file-name' FOR DATABANK transdb;
SQL> CREATE BACKUP IN 'backup-file-name' FOR DATABANK sqldb;
SQL> COMMIT BACKUP;
SQL> EXIT;
```

2 Verify the backup copies from the operating system command line using the DBC program.

```
dbc  backup-file-name  report-filename  sysdb-file-name
.
. (repeat for each backup file created above)
.
```

3 Archive the verified backup copies (e.g. copy to DVD/CD).

The START BACKUP statement will start a backup transaction which will ensure that all the backups taken are consistent with one another (they are effectively backed up at the same point in time).

The CREATE BACKUP statement will only create an empty backup copy file. The entire contents of the specified databank file is copied to the specified file by COMMIT BACKUP. (If any of the backups fail, the ROLLBACK BACKUP statement can be executed to ensure that the log records are preserved.)

The COMMIT BACKUP statement will clear all the LOGDB records that apply to the databanks backed up in the backup transaction.

Databank backup filenames are subject to the same restrictions that apply to the SQL statement CREATE DATABANK - see the *Mimer SQL Reference Manual.*

System Backups Using the Host File System

The procedure for taking databank backups using the host file system is detailed below.

We recommend that you always take a backup of all databanks, including SYSDB, LOGDB, SQLDB and TRANSDB.

Note: The database server must be stopped in order to close the SYSDB databank file for a host system backup. This unlocks SYSDB and ensures that no operations are performed between taking copies of the databanks and dropping the log. However, if using shadowing, databank shadows allow a copy of a databank to be temporarily set offline, e.g. to be backed up, without interrupting normal system use.

To backup databanks using the system backup method:

1. Set the database offline using the following command:

   ```
   SQL> SET DATABASE OFFLINE;
   ```

2. Stop the database server so that the system databanks are closed and can therefore be backed up

   ```
   mimcontrol -t database
   ```

3. Run the DBC program on each databank to verify the physical integrity of the databank files

   ```
   dbc  backup-filename  report-filename  sysdb-filename
   ```

4. Perform the backup, e.g. copy all databank files to tape (including the system databanks SYSDB, LOGDB, TRANSDB and SQLDB).

5. Start the database server

   ```
   mimcontrol -s database
   ```

6. Set the database online again using the following command to clear all log records:

   ```
   SQL> SET DATABASE ONLINE RESET LOG;
   ```

The RESET LOG option removes all records written to LOGDB since the last backup. This is essential to maintain consistency between the log and the backup. If the backup fails, the PRESERVE LOG option should be used when setting the databank online to leave LOGDB unaltered.

It is essential that all databanks are backed up at the same time to ensure logical consistency between them.

It is also important that transactions are in a consistent state which is ensured by using the SET DATABASE OFFLINE statement. The statement will not return until the database has been brought into a consistent state prior to going offline. In particular, setting the database offline will ensure all background processing done by the database server has completed.

Restoring a Databank

Restoring a databank after it has been damaged or destroyed will typically involve both the host file system and SQL statements.

Note: Data need not be restored in the event of a power failure or system shut-down that does not damage the databank files, since any transactions that were committed but not completed at the time of the failure are automatically completed when the databank involved is next accessed.

Any databank restore operation must start with a backup copy of the databank file that is not damaged or corrupt. This is generally the copy taken during the last backup, either taken by the host operating system or by using the SQL system management statements for online backup.

Usually, the host file system is used to copy the backup file from the backup media to disk. The file is generally placed in the normal location for the databank file (as recorded in the data dictionary, `SYSDB`). However, in certain circumstances it may be necessary to place it in an alternative location, e.g. if the disk is unavailable.

The procedure for restoring a databank is as follows:

Note: Step 2 and 3 are only required during certain circumstances:

1. Bring a valid backup copy of the databank from the backup media to disk.
2. If the file has been placed in a location that is different to the location of the original databank file, alter the databank to reference the new file location using the following command:

   ```
   SQL> ALTER DATABANK databank-name INTO 'new-file-name'
   ```

3. If restoring from an older backup, i.e. not the latest one, information should be restored from the `LOGDB` included in the following backup (that was taken after the time the backup restored in step 1 was taken).

   For each `LOGDB` backup file, the information recorded in it should be applied to the databank using the following command:

   ```
   SQL> ALTER DATABANK databank-name RESTORE USING 'logdb-backup-file-name'
   ```

4. Finally, apply the updates made since the most recent backup(s) restored in the preceding steps were taken. These updates are currently recorded in `LOGDB` and they are restored using the following command:

   ```
   SQL> ALTER DATABANK databank-name RESTORE USING LOG
   ```

Restoring SYSDB

If `SYSDB` is lost, a backup copy of `SYSDB` must be restored to allow Mimer SQL to start again. No Mimer SQL-based application can be used before this is done.

If `SYSDB` is lost or corrupted, a backup copy should be copied to the same file location as the original `SYSDB`. The contents of `SYSDB` may then be brought completely up to date by restoring `LOGDB` information. This is done using the backup and restore functionality in the BSQL program.

Start BSQL and login as `SYSADM`, or another user with `BACKUP` privilege. A message is displayed saying that you have an old version of `SYSDB` that must be restored. Answer `Y` to the question `Restore SYSDB` to restore the copy of `SYSDB`.

Since SYSDB always has the LOG option, this will restore SYSDB to the state it had before it was lost.

Example, assuming a backup of SYSDB has been copied to the original location:

```
MIMER/DB fatal error -16159 in function CONNECT
         Old version of the databank SYSDB cannot be accessed without
         restoring the databank with the backup and restore utility

                     -- Restore databank --

Restore SYSDB?[Y]: Y

Databank SYSDB restored from log
```

Re-creating TRANSDB, LOGDB and SQLDB

No Mimer SQL applications can be run if LOGDB, TRANSDB or SQLDB is missing. In this event, starting the BSQL program and logging in as SYSADM will give you an opportunity to re-create the missing databanks with the same filenames as the lost databanks, or to alter the recorded filenames in the case where the physical files were moved.

The following example shows how to re-create LOGDB for a database where this system databank is missing:

```
Mimer SQL command line utility, version 11.0.1A
Username: SYSADM
Password:
2017-06-17 23:15:16.94   <Error>
MIMER/DB kernel error -16142 in function DKOPD1
Databank LOGDB, filename logdb.dbf
File not found, OS error message:
'%SYSTEM-W-NOSUCHFILE, no such file'



-- Redefinition of system databank --

-- Description of databank name and file --

DATABANK
FILENAME
========================================
LOGDB
logdb.dbf

Redefinition of LOGDB? [Y]: y
CREATE new file or ALTER filename for LOGDB? (C/A): c
Size [1000]                  : 5000
Databank LOGDB redefined
```

These databanks (at least TRANSDB and LOGDB) are vital to the system consistency, so we strongly recommend that these files are kept intact whenever possible. A complete backup of the entire database should be made before any system databanks are recreated.

If a database has been operational for some time, a situation may arise where one or more of the system databanks LOGDB, TRANSDB or SQLDB has grown very large. In those cases the ALTER DATABANK DROP FILESIZE statement can be used to shrink the file sizes.

The following sections describe how to re-create each of the respective system databanks.

Creating a New LOGDB

1. Shut down the database server (if not already stopped).
2. Run the DBC tool (Databank Check Utility) on `SYSDB` and all the user databank files to ensure that none are corrupted.
3. Take a valid backup of the whole database.
4. Archive a copy of the `LOGDB` databank file and delete the original file from disk.
5. Start the database server.
6. Start the BSQL program, logging in as `SYSADM`, and when prompted, select the `CREATE` option and specify a path name and a size for the new `LOGDB` databank file.

Creating a New TRANSDB

Note: Do only perform this operation in case of emergency. Important information may be lost and database consistency can not be guaranteed.

1. Shut down the database server (if not already stopped).
2. Ensure that all pending transactions have been flushed to the user-databank files on disk by successful execution of DBOPEN in single-user mode.
3. Archive a copy of the `TRANSDB` databank file and delete the original file from disk.
4. Start the database server.
5. Start the BSQL program, logging in as `SYSADM`, and when prompted, select the `CREATE` option and specify a path name and size for the new `TRANSDB` databank file.

Creating a New SQLDB

1. Shut down the database server (if not already stopped).
2. Delete `SQLDB` from disk.
3. Start the database server.
4. Start the BSQL program, logging in as `SYSADM`, and when prompted, select the `CREATE` option and specify a size for the new `SQLDB` databank file.

Audit trail with READLOG

READLOG is a BSQL function which enables you to read the contents of `LOGDB` so that you can check logged operations performed on the database since the last backup copy backup was taken.

You can use READLOG as an audit trail or, in the event of a system failure, to determine which databanks need to be restored (i.e. which databanks have been altered since the last backup).

See *Mimer SQL User's Manual, Chapter 9, READLOG*.

Chapter 6

Databank Check Functionality

The DBC program allows investigation of databank files to ensure that the physical structure is not damaged. The functionality may also be used to examine the internal organization of a databank file and display statistical information on the databank.

The DBC program traverses every page of a given databank file, and the following are typical tasks that can be examined and verified:

- index and data page consistency
- free page list consistency
- page checksum
- records being in the correct order
- column value accuracy

Note: The functionality checks only the **physical** condition of the databank. Informational errors such as data inconsistency, invalid data format, and data outside the validation limits of domains are not detected by DBC.

DBC - Databank Check

DBC investigates databank files to ensure that the physical structure is not damaged.

Syntax

The overall syntax for the DBC program is:

```
dbc [-a] [-e -s file] [-o file] [databankfile [-f file [-f file...]]]

dbc [--all] [--extended --sysdbfile=file] [--output=file]
    [databankfile [--filename=file [--filename=file...]]]

dbc [-v|--version] | [-?|--help]
```

Command-line Arguments

Unix-style	VMS-style	Function
`-a` `--all`	`/ALL`	If a databank file is not closed, the contents of large objects are by default not checked. The reason is that TRANSDB may contain large object updates which affects the databank state. When `--all` is specified, large objects are checked anyway. The utility can in these cases report errors that will correct themselves after restart of the system. It is therefore recommended to avoid this option.
`-e` `--extended`	`/EXTENDED`	When `extended` is specified, the contents of each row is checked to verify that each column value conforms to the rules for the column data type. SYSDB must be present to do this type of checking.
`-f file` `--filename=file`	`/FILE=file`	Additional filename for multi-file databank. All files for the databank are needed, together with the first databank file given as a separate argument (databankfile). This option is used once for each additional databank file.
`-o file` `-output=file`	`/OUTPUT`	Sequential file created by DBC that contains the result of the verification.
`-s file` `--sysdbfile=file`	`/SYSDBFILE`	Filename for SYSDB used by the databank to check. This filename is required if any tables are using collations. If not specified the correct sort order for such tables is unknown.
`-v` `--version`	`/VERSION`	Display version information.
`-?` `--help`	`/HELP`	Display usage information.
`databankfile`	`databankfile`	Filename for the databank to check.

If the filenames are not specified on the command-line, the program prompts for the name of the databank file, a name for the result file and the name of the system databank (SYSDB) file.

If an error occurs when opening the databank file (e.g. file not found or file locked by another user), or while creating the result file, an appropriate error message is displayed.

If the SYSDB filename is not specified and it is found that tables in the databank use collations, these tables are not verified and a warning message is displayed.

If no errors are detected in the databank file, the following message is shown:

```
No errors found
```

The result file then contains statistics describing the physical databank organization. Otherwise, error descriptions (see below) are written to the result file, and the following message is displayed:

```
* Errors logged in result file
```

The result file should be examined to investigate the nature of the errors, see *Database Consistency* on page *67*.

It should be noted that the DBC program returns an error status (or warning) to the operating system when an error (or warning) is encountered. This may be useful when running it from scripts or in batch mode.

Exit Codes

`DBC` returns a status code to the environment executing the command. The status code can be examined by scripts.

VMS:	On OpenVMS, the status codes correspond to the OpenVMS condition code severity levels. Use the `$SEVERITY` symbol in DCL command procedures.

The following return codes are used:

Linux/Windows	VMS	Usage
`0 (success)`	`1 (success)`	This code is used when the `DBC` command has executed all options with no problems.
`1 (warning)`	`0 (warning)`	Other Error.
`2`	`2 (error)`	Databank error.

Authorization

The DBC program operates directly against the databank file, with no reference to the Mimer SQL database server. The program may not be run on a file which is currently held open (an error message is displayed in such a case). The system administrator should arrange for exclusive access to the databank file during DBC operations.

Result File Contents

Any errors detected in the databank file are written to the result file directly after the identification record for the table affected.

The result file begins with the following information:

- Databank filename.
- Time when the DBC operation was performed.
- Version of Mimer SQL under which the databank was created (if this is not the same as the current version there will also be a ‘converted to’ message).

- A backup timestamp.
- Structure level.
- System identifier for the databank.
- Databank sequence number (0 = master databank, >0 = a shadow).
- Check flag.
- Number of bitmap pages (starting at 0).
- Root pages (starting at 1).
- Number of pages allocated.
- Number of pages used.

If the check flag indicates that the databank was not properly closed when Mimer SQL is stopped, there may be an additional message saying:

```
* No bitmap checking against databank!
```

DBC B*-tree Table Information

An identification record is given for each table in the databank. The following information is shown:

Information	Description
`Tabid`	The system identification number of the table. This is the number used to identify the table in the data dictionary. The table name is not stored directly in the databank file.
`Startp`	The page number of the start page for the highest index level. If the number of levels is 1, this is the only data page. If the start page is 0, the table is empty.
`Levels`	The number of levels in the table storage structure.
`Keylen`	The length of the primary key in bytes.
`Reclen`	The record length (row length) of the table.

`Type of table`	Any of the following values:
	`Base table`
	`Secondary index table`
	`Collation description table`
	`LOB directory table.`

`Type of compression`	Any of the following values:
	`Mimer LZM compression`
	`Mimer LZU compression`
	`Mimer RLE compression`
	`none.`

`Collation version`	Displayed if any key part of the table is using collation.
`Status of table`	`Resident` or `Marked for delete`.
`Index page size`	Size of the index page in bytes.

Information	Description
`Data page size`	Size of the data page in bytes.
`Number of index pages`	Indicates how many index pages were checked.
`Number of data pages`	Indicates how many data pages were checked.
`Required/Allocated datapages`	Gives an approximate measure of the space used in the data pages. This is expressed as a percentage, which may be >100% for data pages having variable-length (i.e. compressed) records because the required number of pages is calculated based on uncompressed record sizes.
`Reached no of records`	If no errors are reported, the number of records reached is equal to the total number of records (rows) stored for the table.

DBC Sequential Table Information

The following information is shown:

Information	Description
	Table name.
`Tabid`	Ordered identification.
`Startpage`	First page in the sequential file.
`Endpage`	Last page in the sequential file.

`Type of table`	Any of the following values:
	`Sequential table`
	`Collation description table`
	`LOB directory table`.

`Type of compression`	Any of the following values:
	`Mimer LZM compression`
	`Mimer LZU compression`
	`Mimer RLE compression`
	`none`.

`Status of table`	`Resident` or `Marked for delete`.
`Index Page size`	Size of the index page in bytes.
`Data Page size`	Size of the data page in bytes.
`No. of index pages read`	Number of index pages checked.
`No. of data pages read`	Number of data pages checked.
`No. of records read`	Number of records checked.

DBC LOGDB Backup Information

The following information is shown:

Information	Description
`Timestamp page`	Describes actual timestamps for databanks included in the log.

Error Messages

The following error situations are described by explicit messages in the result file:

Bitmap Errors

Error message	Explanation
`* Illegal number of free bits in bitmap`	The number of free bits in the bitmap is marked as a negative number or as a number greater than the permissible value.
`* Illegal pointer to first word with free bit in bitmap`	The pointer to the first word is either negative or greater than the number of words per page, or points outside the number of allocated pages.

Root Page Errors

Error message	Explanation
`* Illegal record length in root page`	The record length is not valid for the current version or for the site.
`* Pageno. outside databank: x` `* Pageno. not marked as used: x`	The reference to the page number (`x`) is invalid. (Applies only where there is more than one root page.)

Sequential Structure Errors

Error message	Explanation
`* Pointer to previous page invalid`	Error in page linkage.
`* Invalid record length`	The record length is either not the same as in the previous page or outside the page limits.
`* Record crosses page boundary`	A record stretches over the page limits.

Table Structure Errors

Error message	Explanation
`* Error in root record`	The value for start page, levels, key length or record length is outside the legal values for the site.
`* Page has illegal record length: x`	The record length (x) given in the page is not the same as that given for the table.
`* Page has illegal last-record pointer: x`	The pointer (x) to data within a page is outside the page limits. The limits are the values of bytes/header and bytes/page.
`* Page has records in wrong order. Pos: x`	The records in the page are not correctly sorted. The value of x is the byte position within the page for the start of the wrong record.
`* Page has last-record key > index key. Pos: x`	The records within a page include key values greater than those in the index level above. The value of x is the byte position within the page for the start of the last record.
`* Page no. outside databank: x`	Reference is made to a page number (x) which is higher than the highest allocated page.
`* Page no. referenced twice`	There are at least two references to the same page number (x). One of these references should also give another error, or an error should have been notified earlier in the file.
`* Page no. not marked as used: x` `Bitmap pageno: Word: Bit:`	A page that is used is illegally marked as free. Continued insertion of data in the databank will result in a double referenced page.

All table structure errors are followed by a listing of the page numbers passed in the B*-tree structure on the way to the error. In the example below a y-value indicates the byte position in the page (corresponding x-value) where the record holding the reference to the next level starts:

```
B*-tree
Page no:    x1   x2   x3  ....  xn
Byte pos:   y1   y2   y3  ....  yn
```

If the error occurs at an index level, the following additional message is given, and no checks are made at lower levels:

```
Branch is interrupted
```

Internal Databank Check

In addition to the DBC program there is an internal databank check feature within the database server. When the database server is started, if it is noticed that the system was closed in a non-proper way, this verification will be triggered. The default check includes page consistency and checksum validation. If an error is found, the broken databank file will not be opened and the database will not be fully functional. In a situation like this a backup, or corresponding, must be used to get the system operational.

On VMS and Linux the internal databank check feature is controlled using the database server configuration parameter `DBCheck`, see *MULTIDEFS Parameters* on page *163*.

On Windows the Mimer Administrator is used to control the internal databank check feature.

Chapter 7

DBOPEN - Databank Open

The DBOPEN program is used to make sure that the users can open all databanks quickly.

It can take relatively longer to open a databank if it is a large databank that was closed abnormally (for example if the machine crashed), because a databank check is automatically initiated following such a situation.

DBOPEN - Databank Open functionality

Multi-user Mode

When DBOPEN is run in multi-user mode, the index pages of the databank are checked before databank access can proceed, but the bulk of the databank contents (contained in the data pages) is checked in the background, thus minimizing the delay.

Background Threads

The checking performed by DBOPEN is done by the background threads, see *Number of Background Threads* on page *47*. Therefore, increasing the number of background threads will increase the efficiency of the checking.

DBOPEN should normally be run as soon as the database server has been started.

Syntax

The overall syntax for DBOPEN is:

```
dbopen [-s|-m] [-u user] [-p pass] database

dbopen [--single|--multi] [--username=user] [--password=pass] database

dbopen [-v|--version] | [-?|--help]
```

Command-line Arguments

Unix-style	VMS-style	Function
-m --multi	/MULTI	Connects to the database in multi-user mode.

Unix-style	VMS-style	Function
`-p password` `--password=password`	`/PASSWORD=password`	Specifies the password used in connecting to Mimer. If the switch is omitted the user is prompted for a password, unless OS_USER is specified with the username switch, as described above. **VMS:** Note that in an Open VMS environment it might be necessary to enclose the password in quotation marks as the value otherwise is translated to upper case.
`-s` `--single`	`/SINGLE`	Connects to the database in single-user mode.
`-u username` `--username=username`	`/USERNAME=username`	Specifies the username used when connecting to Mimer. To connect using OS_USER login, give `-u ""`, `--username=""`, or `/USERNAME=""`.
`database`	`database`	Specifies the name of the database to access.
`-?` `--help`	`/HELP`	Show help text.

If the optional database name is not specified, the default database will be accessed, see *The Default Database* on page *37*.

If the username or password arguments are omitted, the program will prompt for these values.

If neither -s nor -m is specified for the optional mode flag, the way the database is accessed will be determined by the setting of the MIMER_MODE variable, see *Specifying Single-user Mode Access* on page *151*, or, if this is not set, it will be accessed in multi-user mode.

Linux:	The Unix-style command-line flags must be used on a Linux machine.

VMS:	Either the Unix-style or the VMS-style command-line flags may be used on an OpenVMS machine - see the *Mimer SQL VMS Guide* for more details.

Win:	The Unix-style command-line flags can be used from a Command Prompt window.

Exit Codes

The DBOPEN program returns an error status to the operating system when an error is encountered. This may be useful when running it from scripts or in batch mode.

Functions

The DBOPEN functionality opens all available databanks in a system.

As each databank is opened, the integrity is checked and any transactions that were interrupted by the abnormal close are completed. (The integrity check is always performed when opening a databank that has not been closed normally.)

The checks performed when an abnormally closed databank is opened may take some time, particularly if the databank is large. Running DBOPEN means that the checks are performed at a time determined by the system administrator, rather than a time determined by application programs. As a result, users will always have fast access to all databanks.

The databanks are opened in a randomly determined order. Running several DBOPEN sessions in parallel may speed up the checking process for the database as a whole.

Authorization

Any user who has SELECT access to the data dictionary table SYSTEM.DATABANKS can run DBOPEN.

Output Example

The following DBOPEN example shows the output from running dbopen after the system was shut down improperly:

```
dbopen -s -usysadm -p******

MIMER Databank Open Utility
Version 11.0.1A

2017-09-15 14:19:12.22   <Information>
MIMER/DB kernel error -16211 in function DKVED0
Databank SYSDB not properly closed, dbcheck initiated

2017-09-15 14:19:22.67   <Information>
MIMER/DB kernel error -16211 in function DKVED0
Databank TRANSDB not properly closed, dbcheck initiated

2017-09-15 14:19:22.71   <Information>
MIMER/DB kernel error -16211 in function DKVED0
Databank LOGDB not properly closed, dbcheck initiated

Opening databank APPDB1

All user databanks opened without errors
```

Chapter 8

Loading and Unloading Data and Definitions

Mimer SQL provides you with flexible methods for loading information to and from databases using the `LOAD` and `UNLOAD` commands and the `MIMLOAD` program.

`LOAD` and `UNLOAD` can be used with Mimer SQL from any ODBC based SQL command interpreter.

Using the `LOAD` command, you can:

- load information from one or more files into a Mimer SQL database
- optimize loading for best performance
- select exactly the information you want and place it precisely where you want it using an SQL statement
- log the load operation.

Using the `UNLOAD` command, you can:

- customize the information – unload whole databanks or select specific information using an SQL statement
- log the unloading operation.

Using the `MIMLOAD` program, you can use `LOAD` and `UNLOAD` directly from your operating system command prompt. Using the `STDIN`, `STDOUT` and `STDERR` options in the `LOAD/UNLOAD` syntax, you can enable command line file redirection for input, output and logging.

MIMLOAD - Data Load and Unload

MIMLOAD enable you to use the LOAD and UNLOAD commands at your operating system's command prompt.

Syntax

You control MIMLOAD using flagged information specified on the command-line.

The overall syntax for MIMLOAD (expressed in short form Unix-style) is:

```
mimload [-s] [-u user] [-p pass] [-e prog] [-i pass]] command [database]

mimload [--single] [--username=user] [--password=pass]
    [--program=prog] [--using=pass]] command [database]

mimload [-v|--version] | [-?|--help]
```

Note: If you are using double quotes in the LOAD/UNLOAD statement, they must be escaped using the back-slash character \ when Unix-style switches are used. For VMS-style switches, a double quote " is used as escape character.

Command-line Arguments

You can use the following arguments with MIMLOAD.

Unix/Windows-style	VMS-style	Function
-e program --program=program	/PROGRAM=program	Specifies an optional program ident, to be entered in the connection phase. This can be specified more than once and thus allows entering deeper entry levels.
-i password --using=password	/USING=password	Password for the program ident specified by the preceding --program switch.
-p password --password=password	/PASSWORD=password	Specifies the password for the user. When connecting using an OS_USER login, no password is needed. Note that in an Open VMS environment it might be necessary to enclose the password in quotation marks as the value otherwise is translated to upper case.

Unix/Windows-style	VMS-style	Function
`-u user` `--username=username`	`/USERNAME=username`	Specifies the username used when connecting to Mimer. If not specified, OS_USER login is assumed.
`command`	`command`	The `LOAD`/`UNLOAD` statement to be used for data management. Enclose in double quotes. See *LOAD - Loading Data* on page *98* and *UNLOAD - Unloading Data* on page *101* for details.
`database`	`database`	Specifies the name of the database to access. If specified, it must be the last argument. If you do not specify a database name, the default database will be used.
`-?` `--help`	`/HELP`	Show help text.

Exit Codes

The following error codes are used:

Linux/Windows	OpenVMS	Usage
`0 (success)`	`1 (success)`	This code is used when `MIMLOAD` has executed successfully.
`> 1 (error)`	`2 (error)`	This error code is used when `MIMLOAD` failed to execute successfully.

Examples

Loading Data

The following example loads the file `store.sql` into the default database.

Unix/Windows-style

```
mimload --username=user1 --password=UsrPwd "load from 'store.sql' log stderr"
```

VMS-style

```
MIMLOAD /USERNAME=USER1 /PASSWORD="UsrPwd" "LOAD FROM 'STORE.SQL' LOG STDERR"
```

Unloading Data

'The following example unloads all the definitions and data owned by the user store_adm from the database store to the file store.sql.

Unix/Windows-style

```
mimload -u store_adm -p StrPwd "unload to 'store.sql' from current user" store
```

VMS-style

```
MIMLOAD /USERNAME=STORE_ADM /PASSWORD="StrPwd" "UNLOAD TO 'STORE.SQL' FROM
CURRENT USER" STORE
```

Using STDIN/STDOUT/STDERR

STDIN, STDOUT and STDERR are short names for the standard input, standard output and standard error streams, respectively.

Windows/Unix-style Examples

In the commands below, STDIN is denoted by '<' (this is short for '0<', equal to reading from file descriptor number 0 which is the standard input). STOUT is denoted by '>' (this is short for '1>', equal to writing to file descriptor number 1 which is the standard output) and STDERR is denoted by '2>' (equal to writing to file descriptor number 2 which is the standard error).

In the following example, the generated output from UNLOAD is written to standard output, which in this case is redirected to the file store_db.unl. The log information is written to standard error, which is redirected to the file store_db.log.

```
mimload -u store_adm -p StrPwd "unload to stdout log stderr from databank
store_db" store > store_db.unl 2> store_db.log
```

In the following example, input to LOAD is taken from standard input, i.e. from the file store_db.unl, and log information is written to standard error, in this case to the file store_db_load.log.

```
mimload -u store_adm -p StrPwd "load from stdin log stderr" store <
store_db.unl 2> store_db_load.log
```

VMS-style Examples

In the OpenVMS environment, you can use STDIN, STDOUT and STDERR in two ways, either by defining SYS$INPUT, SYS$OUTPUT and SYS$ERROR or by using pipe mode which gives the Linux behavior for use of file redirection. Both methods are shown below.

In the following example, the generated output from UNLOAD is written to standard output, which in this case is redirected to the file D1:<DATA>STORE_DB.UNL. The log information is written to standard error, which is redirected to the file D1:<LOG>STORE_DB.LOG.

Example using SYS$OUTPUT and SYS$ERROR:

```
DEFINE/USER SYS$ERROR D1:<LOG>STORE_DB.LOG
DEFINE/USER SYS$OUTPUT D1:<DATA>STORE_DB.UNL
MIMLOAD /USERNAME=STORE_ADM /PASSWORD="StrPwd" "UNLOAD TO STDOUT LOG STDERR
FROM DATABANK STORE_DB" STORE
```

Example using pipe mode:

```
PIPE MIMLOAD /USERNAME=STORE_ADM /PASSWORD="StrPwd" "UNLOAD TO STDOUT LOG
STDERR FROM DATABANK STORE_DB" STORE > D1:<DATA>STORE_DB.UNL 2>
D1:<LOG>STORE_DB.LOG
```

In the following example, input to `LOAD` is taken from standard input, i.e. from the file `D1:<DATA>STORE_DB.UNL`, and log information is written to standard error, in this case to the file `D1:<LOG>STORE_DB_LOAD.LOG`.

Example using `SYS$OUTPUT` and `SYS$ERROR`:

```
DEFINE/USER SYS$ERROR D1:<LOG>STORE_DB_LOAD.LOG
DEFINE/USER SYS$INPUT D1:<DATA>STORE_DB.UNL
MIMLOAD /USERNAME=STORE_ADM /PASSWORD="StrPwd" "LOAD FROM STDIN LOG STDERR"
STORE
```

Example using pipe mode:

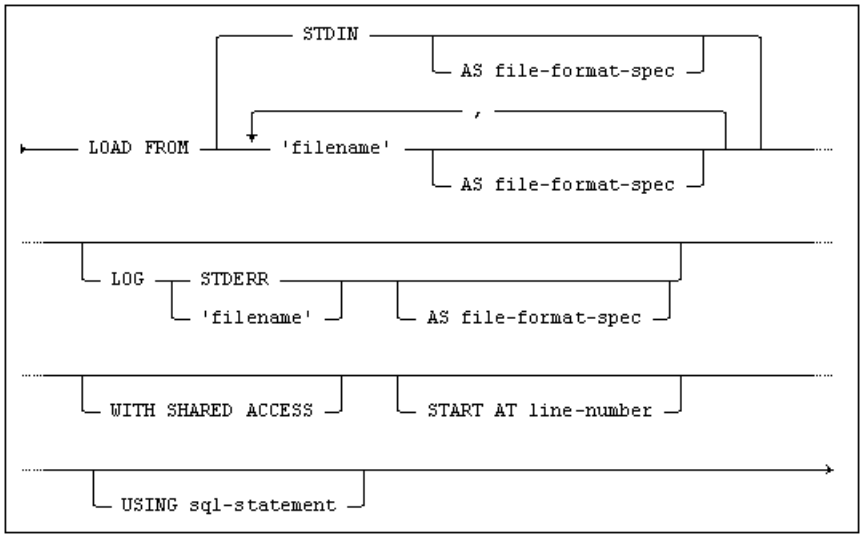
```
PIPE MIMLOAD /USERNAME=STORE_ADM /PASSWORD="StrPwd" "LOAD FROM STDIN LOG
STDERR" STORE < D1:<DATA>STORE_DB.UNL 2> D1:<LOG>STORE_DB_LOAD.LOG
```

LOAD - Loading Data

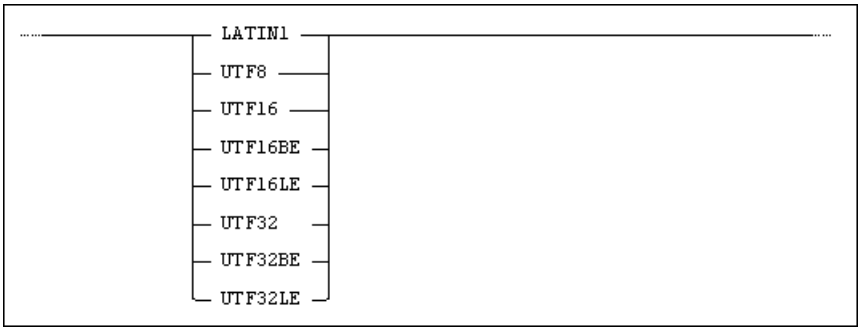
You load definitions and/or data into a Mimer SQL database using the `LOAD` statement.

Syntax

The `LOAD` command has the following syntax:



where `file-format-spec` is:



Usage

`MIMLOAD`, or with any ODBC-based SQL interpreter.

For information on `MIMLOAD`, see *MIMLOAD - Data Load and Unload* on page *94*.

Description

The `LOAD` command copies definitions and/or data from one or more files. When loading information from more than one file, the files are read in the order defined. The input file(s) are expected to form a valid sequence of definitions, data descriptions and data.

Triggers defined against the affected tables are applied when the data is loaded.

About Files

When a file contains data for more than one table, the data for each table must be contained in a section that is introduced by a data description header. For more information, see *Data Description Headers and Files* on page *104*.

If the data in the file does not have a data description header, there must be a data description file that contains the header information.

This means that the file can only contain data from one table. Data description files and data files can of course be concatenated into a single file containing data for several tables.

A definition file contains definition statements to create objects. A definition file can, for example, be divided into two files where one file is place first in the file list; i.e. executed before any data is loaded, and the other file is placed at the end of the file list, i.e. executed after the data is loaded.

An example file sequence can be as follows: first in the file list, a file that contains object definitions; second, a file that describes the data to be loaded (the information in this file is equal to the corresponding information that can be given as a header in the data file); third, the data file; and fourth, a second definition file including referential constraints and triggers.

You can specify the name of the table into which the information shall be loaded in the data file header(s) or the data description file. The default is the table name from which the data was unloaded.

When `LOAD` scans a file, it detects if a field uses a text qualifier by checking if the first character in the field is a text qualifier. If a text qualifier character is found in the field data, the character is doubled, i.e if the text qualifier is a double quote, the data `ab'c` is equal to the data `'ab''c'`.

The STDIN Option

When you use the `STDIN` option, input is read from the standard input stream. See *Using STDIN/STDOUT/STDERR* on page *96*.

The AS Option

By using the `AS` option together with a filename specification, you can specify the character set used by the file to be loaded.

The character sets available are: `LATIN1`, `UTF8`, `UTF16`, `UTF16BE`, `UTF16LE`, `UTF32`, `UTF32BE` and `UTF32LE`.

`UTFxxBE` and `UTFxxLE` means `UTFxx` format with big or little endian byte order. `UTFxx` without endian notion means that the common endian for the current platform is assumed.

The default character set used, if you do not use the `AS` option, is the default used in your host operating system.

For more information, see *File Format Specifications* on page *106*.

LOG

You can specify the log file using `LOG`. This log file will include warnings and progress information about the load operation. If you do not use the `LOG` option, logging will be written to the standard error stream.

The STDERR Option

When you specify `LOG STDERR`, informational messages are written to the standard error stream. See *Using STDIN/STDOUT/STDERR* on page *96*.

The WITH SHARED ACCESS Option

LOAD's default behavior implies that you have exclusive access to the databank being loaded with data. If you need shared access, you can use the WITH SHARED ACCESS option. In most cases, this will lead to a slower data load using row-wise insert.

By default, LOAD uses a fast data load facility to increase performance. The alternative is to insert data row-wise as if using an SQL INSERT statement.

LOAD uses the fast data load facility in most cases, but there are some situations that need row-wise insertion due to certain referential constraints. In such cases, a warning message will tell you that fast data load cannot be used and the operation will continue using row-wise insertions.

When row-wise insertions are performed, loads are recorded in LOGDB (assuming the databank is defined with the LOG option).

The START AT Option

You can use the START AT option to restart a failed load operation.

The START AT value can be set to a line where a data definition statement is located or a data descriptor header starts (#data).

The USING Option

The USING option enables you to use an SQL statement to specify the information to be loaded and the target for the information.

The SQL statements you can use are: INSERT, UPDATE, DELETE or CALL to a procedure with input parameter markers only.

Examples

The following example is a straightforward import of the input file, using default options:

```
LOAD FROM 'table_t.data' LOG 'table_t.log';
```

The following example imports a data file, preceded by a data description file, using the default options:

```
LOAD FROM 'table_t.desc', 'table_t.data' LOG 'table_t.log';
```

The following example imports the first four columns of data in the file to the table named details from a file in UTF16 format:

```
LOAD FROM 'table_t.data' AS UTF16
   LOG 'table_t_dataload.log'
   USING INSERT INTO details VALUES (?,?,?,?);
```

The following example uses an UPDATE statement where the first column C1 and the second column C2 of the data input file are used:

```
LOAD FROM 'table_t.data' AS UTF16
   LOG 'table_t_dataload.log'
   USING UPDATE details SET c1=? WHERE c2=?;
```

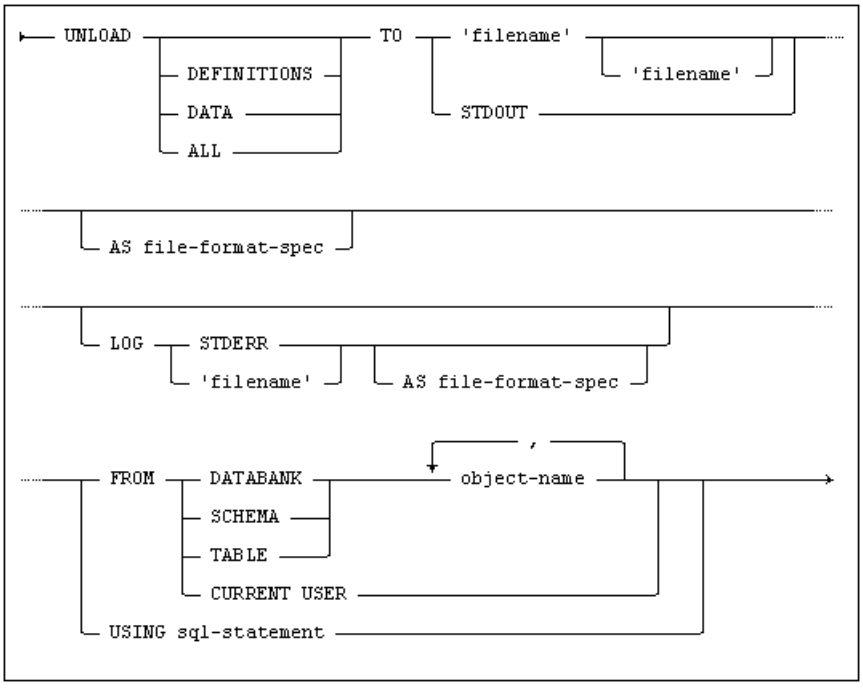
The following example uses a DELETE statement where the input data is used to qualify records to delete:

```
LOAD FROM 'table_t.data' AS UTF16
   LOG 'table_t_dataload.log'
   USING DELETE FROM details WHERE c2=? AND c3=?;
```

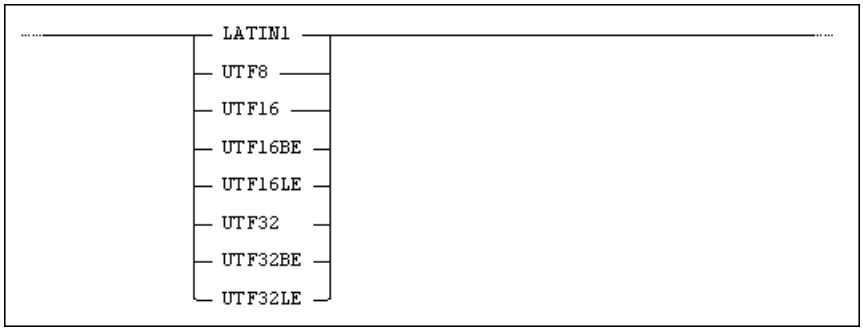
UNLOAD - Unloading Data

You use the UNLOAD command to unload data and/or definitions from a Mimer SQL database to a file.

Syntax



where file-format-spec is:



Usage

Any ODBC-based SQL interpreter or with the MIMLOAD program.

For information on MIMLOAD, see *MIMLOAD - Data Load and Unload* on page *94*.

Description

The UNLOAD command generates data and/or definitions and places the result in a file.

If one file is specified in the UNLOAD command, both data and definitions will be placed in that file. If two files are specified, definitions will be placed in the first file, and data in the second file. (This makes it easier to change the table name before creating and loading the table.)

You can use the options ALL (default), DEFINITIONS or DATA to specify the information you want.

When generating the data and definitions, a data description header is created before information is written to the file. If information from several tables is generated, a data description header for each data section is created.

A data description header contains escaping information and column separator information. For more information, see *Data Description Headers and Files* on page *104*.

Data Escape Mode

UNLOAD generates data in escaped mode. This means that the data description header includes the data escape mode option.

When using data escape mode, the following characteristics are enabled, from the UNLOAD perspective:

- Data from a specific table is ended by the escape sequence '_' to mark end-of-table.
- Null values are indicated by the escape sequence '\-'.
- BLOB and BINARY columns are unloaded in HEX code with a leading '\x' escape sequence for each byte.
- BLOB, CLOB and NCLOB columns are unloaded so that the value length is given in front of the value as in the following CLOB example: '11:Abracadabra'
- For CHAR, NCHAR, CLOB and NCLOB columns, the escape sequence '\x' is used only when there is binary data, such as ISO control codes, new-line characters, etc. in the data.
- The '\u' escape sequence is used only when Unicode data is to be written to Latin1 files.

For information on escape sequences, see *Escape Character Sequences* on page *105*.

The STDOUT Option

When you use the STDOUT option, generated output is written to the standard output stream. See *Using STDIN/STDOUT/STDERR* on page *96*.

The AS Option

By using the AS option together with a filename specification, you can select the character set of the generated file. You can choose: LATIN1, UTF8, UTF16, UTF16BE, UTF16LE, UTF32, UTF32BE or UTF32LE.

UTFxxBE and UTFxxLE means UTFxx format with big or little endian byte order. UTFxx without endian notion means that the common endian for the current platform is assumed.

The character set used, if you do not use the AS option, is UTF8 with BOM.

For more information, see *File Format Specifications* on page *106*.

The LOG Option

You can generate a log of the operation using the LOG option. The log file will include warnings and progress information about the operation. If you do not use the LOG option, logging will be written to the standard error stream.

The STDERR Option

When you use the LOG STDERR option, informational messages are written to the standard error stream. See *Using STDIN/STDOUT/STDERR* on page *96*.

The USING and FROM Options

To specify the information to be unloaded, you use the USING or FROM options.

With the USING option, an SQL statement, such as SELECT * FROM T1; or a CALL to a procedure with parameter markers (?) for output parameters only, can be used to specify the source.

By using an SQL statement to form the source for the export operation, there are many possibilities available to format and customize the output.

With the FROM option, one or several databanks, tables or schemas can be used to form the source for the export operation. If using the FROM CURRENT USER option, the current ident is exported.

If tables are joined in the SQL statement used, and definitions are generated, a new table that is a reflection of the result of the join is defined. The default name of the new table is table1.

Error Management

The UNLOAD command runs until a major error is encountered. Minor problems are reported as warnings if LOG is enabled. If a fatal error occurs, an error message is displayed and the operation is aborted.

Examples

The following example will export the table details, with all related definitions, to a file:

```
UNLOAD DEFINITIONS TO 'table_t.def' FROM TABLE details;
```

The following example will export the CREATE statement for table details together with all data in the table to a file in UTF16 format. A log file is used:

```
UNLOAD TO 'table_t.all' AS UTF16
   LOG 'table_t.log'
   USING SELECT * FROM details;
```

The following example will export the CREATE statement for table details to the definitions file createtable.dat, and its data to another file tabledata.dat:

```
UNLOAD ALL TO 'createtable.dat', 'tabledata.dat' FROM TABLE details;
```

Data Description Headers and Files

Data description headers and files are used to describe the data that follows.

The following table describes data description header elements:

Element	Usage	Description
`#data`	Required	Data description header start identifier.
`escape mode`	Optional	Indicates that the data is escaped, i.e. that some elements of the data are tagged for secure recognition at `LOAD`. See the table below. When using `UNLOAD`, data escape mode is always used.
`column separator '`*x*`'`	Optional	Indicates which character is the column separator when reading the data. The default is the comma character (`,`). If this option is not used, `LOAD` assumes that the comma character is the column separator.
`text qualifier '`*x*`'`	Optional	Indicates which character is the qualifier for text strings in the data. The default is the double quote character (`"`). If this option is not used, `LOAD` assumes the double quote character as the text qualifier or unqualified data.
`null indicator '`*x*`'`	Optional	Indicates which character is the null value if found in a data field. If this option is not stated, `LOAD` assumes the empty string, i.e. two consecutive column separators, as a null value. In data escape mode, `'\-'` is treated as the null value.
`using` *insert-statement*	Optional	The SQL `INSERT` statement that indicates where, and in what way, data should be loaded. This statement is used in the situation where the `LOAD` statement itself does not include a `USING` clause.
;	Required	Data description header end identifier.

As shown in the table above, the characters used to specify column separators, text qualifiers and null indicators must be enclosed in single quotes. If you use a single quote to specify a column separator, text qualifier or null indicator, you must enter it twice, for example, you would specify a single quote as a column separator as `''''`.

Data Description Header Examples

For data unloaded from a Mimer SQL database using `UNLOAD`, the data description header generated could look as follows:

```
#data escape mode using insert into t (c) values (?);
```

The example above implies the following for `LOAD`:

- The column separator is the comma character (default).
- Text strings are presumed to be unqualified or qualified with the double quote character.
- Data escaping is assumed (see the table below).
- The `USING` statement in the header will be used if no `USING` clause is given in the `LOAD` statement.

The following is another example of a data description header where all optional elements mentioned above, except `data escape mode`, are used:

```
#data
column separator ':'
text qualifier '!'
null indicator '§'
using insert into t1 (c1,c2,c3) values (?,?,?);
```

In the example above, the table `t1` and the columns `c1`, `c2` and `c3` are supposed to exist when starting the data load. Specific characters for `column separator`, `text qualifier` and `null indicator` are defined.

Escape Character Sequences

If data escape mode is specified, the back-slash character (`\`) is used as the escape character. The character following the escape character can be one of '`x`', '`u`', '`-`' or '`_`'. See the following table for a description of valid escape character sequences:

Escape character sequence	Usage Description	Example
`\x` (lower case letter '`x`')	Preceding a hexadecimal byte value. A HEX value is assumed to be two HEX value digits, i.e. 0-F.	`\x1A`
`\u` (lower case letter '`u`')	Preceding a unicode value. A Unicode value is assumed to be eight HEX value digits, i.e. 0-F.	`\u12345678`
`\-` (dash)	Null value	`\-`
`\_` (underscore)	End of table, including end of stream or file	`\_`

Note: If you do not use `data escape mode`, end of file is treated as end of table. This means that such a data file only can contain data for one table.

File Format Specifications

The default `UNLOAD` file format is `utf8`, which `LOAD` handles without having to specify it. This means that in most cases you don't need to specify the file format, if you don't do it at `UNLOAD` you don't have to do it at `LOAD` either.

The various file formats that can be used are briefly described in the following table:

File Format	Description
`latin1`	ISO 8859-1, i.e. ISO's 8-bit single-byte coded graphic character set for Western languages.

`utf8`
`utf16`
`utf32`
	Unicode Transformation Formats, standard character encoding schemes in accordance with ISO 10646.
	For more information, see https://www.unicode.org

`utf16be`	`UTF16` format with big endian byte order.
`utf16le`	`UTF16` format with little endian byte order.
`utf32be`	`UTF32` format with big endian byte order.
`utf32le`	`UTF32` format with little endian byte order.

Chapter 9

Replication

Mimer SQL supports continuous replication of data from a source database to a target database. The replication is based on triggers that store changes for replicated tables in log tables. The replication system reads these log tables and carries out the operations on the target system. The replication system consists of three separate programs, MIMREPADM, REPSERVER and MIMSYNC.

MIMREPADM is a command line based tool for defining which tables that should be replicated. It is also used for install and uninstall of the replication dictionary, which is needed for maintaining information about replicated tables.

REPSERVER handles the actual replication.

The synchronization program, MIMSYNC, performs data manipulation operations to ensure that source and target tables contain the same rows.

Note: The replication system cannot be used to create tables on the target database. It is the responsibility of the user to create these tables.

Requirements

These programs use ODBC for database access. The Mimer SQL version of the source database must be 9.3.5 or later. The Mimer SQL version of the target database must be 9.2.4 or later.

There must be a Mimer license key that allows replication on the source database. No replication license key is required on the target database.

Restrictions

For a replicated table, the value of a primary key column must not be updated.

Note: If a primary key value is updated, that row will not be replicated. In that case mimsync can be used to correct the replicated data. See *MIMSYNC - Synchronizing Tables* on page *119*.

MIMREPADM - Replication Administration

This section describes how to set up a replication environment.

Syntax

The following options can be specified as command line arguments for the `MIMREPADM` program:

```
mimrepadm -i value|--install=value [source_database]

mimrepadm -u value|--uninstall=value [source_database]

mimrepadm -r pass|--rpassword=pass
    [--susername=user] [--spassword=pass]
    [--tusername=user] [--tpassword=pass]
    [--tdatabase=name] [source_database]

mimrepadm [-v|--version] | [-?|--help]
```

Command-line Arguments

You can use the following arguments with MIMREPADM.

Unix/Windows-style	Function
-i value --install=value	Create the source or target replication dictionary. Valid values are: source target
-r password --rpassword=password	REPADM password.
-u value --uninstall=value	Drop the source or target replication dictionary. Valid values are: source target
-u user --username=username	Specifies the username used when connecting to Mimer. If not specified, OS_USER login is assumed.
-v --version	Display version information.
database	Specifies the name of the database to access. If specified, it must be the last argument. If you do not specify a database name, the default database will be used.
-? --help	Show help text.
--susername=user	Source username.
--spassword=pass	Source password.
--tdatabase=name	Target database.
--tusername=user	Target username.
--tpassword=pass	Target password.

Replication Setup

The first step in setting up a replication environment is to use the install option of the MIMREPADM program. This needs to be done on both the source database and the target database. This is done by running the MIMREPADM program with the following arguments

```
mimrepadm --install=source [source-database-name]
```

and

```
mimrepadm --install=target [target-database-name]
```

If the database name is omitted, the program will prompt for a database name. The installations must be run as a user with DATABANK and IDENT privilege. The program will prompt for name and password for such a user.

The installation on the source database consists in creating two users, REPADM and REP_SOURCE_USER. REPADM is the user that will own the replication dictionary. These dictionary tables are created in a databank named REPADM that is created by the install program. REP_SOURCE_USER is the user, which will be used when performing the actual replication on the source database.

On the target database the installation consists in creating the user REP_TARGET_USER. This user is used for performing operations on replicated tables on the target database.

The replication environment can be removed by using the uninstall option for the MIMREPADM program, i.e.

```
mimrepadm --uninstall=source [source-database-name]
```

and

```
mimrepadm --uninstall=target [target-database-name]
```

Uninstall is not allowed if there are any subscriptions (see next chapter) defined for the database.

Note: Do not try to remove the replication environment by dropping any users explicitly. Always use the uninstall option for this.

Replication Administration

A subscription defines which tables that should be replicated for a specific source database and target database. It is possible to have multiple subscriptions between the same source and target database, e.g. if tables owned by different users should be replicated. Subscriptions are defined by using the MIMREPADM program. In this case, the MIMREPADM program is started with the following options specified

```
mimrepadm -r password  database
```

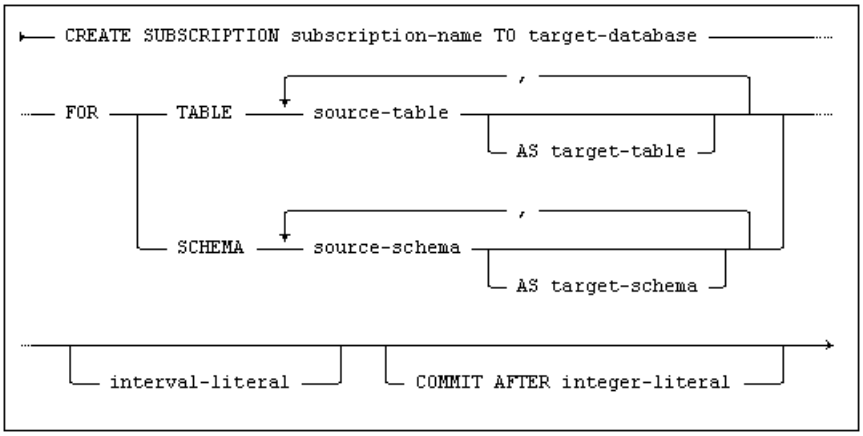
where password is the password for the REPADM user and database is the name of the source database. Before creating a subscription it is required to specify the target database and the users, which own the tables to be replicated. These values can be specified by giving arguments to the MIMREPADM program or by using the CONNECT SOURCE USER and CONNECT TARGET USER statements.

When the MIMREPADM program is started, it will prompt for commands. Since the program reads from standard input and writes to standard output it is possible to use OS primitives for piping and redirection. The following sections describe the available commands.

As a step in the setup process, the target database environment (users, databanks and tables) must be created. This must be done manually, since the replication system cannot be used to any target database objects. (Perhaps MIMLOAD can be useful.)

CREATE SUBSCRIPTION

Creates a subscription.



Description

The name of a subscription follows the normal rules for identifiers in SQL (see *Mimer SQL Reference Manual* for more details.) When a subscription is created, a log table and triggers for logging all write operations are created for each table in the subscription. Since it is only the owner of a table that has the right to create triggers on a table, all tables in a subscription must be owned by the source user. Further requirements is that there is a primary key constraint defined for the replicated table and that the table is not located in a databank having work option. The table used for logging all changes done on the replicated table is created in the same databank in which the replicated table is located. The logging will include information about which transaction the operation belongs to.

When a subscription is created, a corresponding table for each replicated table must exist on the target database. They do not need to have the same name or be located in the same schema as on the source database but the definition must be the same. Within a subscription it is possible to define that a table is replicated to multiple tables on the target database.

Note: There is currently no option for automatically creating the tables in the target database. It is the responsibility of the user to create these tables.

Since the `MIMREPADM` program grants delete, insert and update privilege on the specified tables to the `REP_TARGET_USER` user, the target user must have these privileges with grant option for all tables in a subscription.

Example

```
REPLICATION>create subscription SUB_MIMER_STORE to DUSTPUPPY
REPLICATION& for schema MIMER_STORE as ROC
REPLICATION& interval '10' minute
REPLICATION& commit after 10;
```

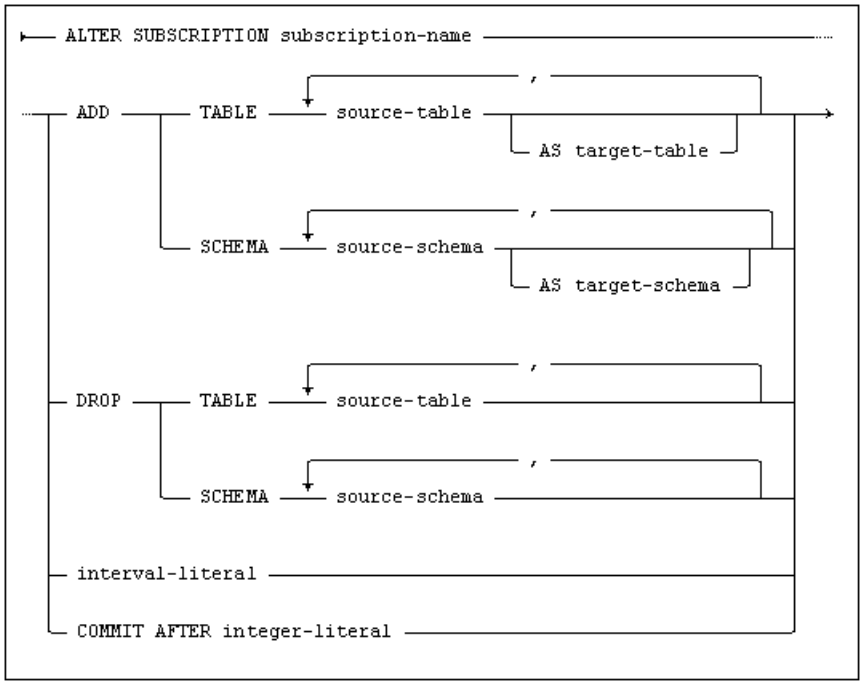
This means that all tables in the schema MIMER_STORE will be replicated to similarly named tables in the schema ROC on the database DUSTPUPPY. The interval value specifies at which interval the `REPSERVER` program will look for data to replicate. The interval literal must be a short interval, i.e. date fields from day to second can be used. (The interval literal format is described in *Mimer SQL Reference Manual*). The value cannot be negative. The default value is 15 minutes. If a zero interval is given, the replication will be continuous. The commit after clause tells how many source transactions should be bundled in a target transaction. The default value is 1.

```
REPLICATION>create subscription SUB_MIMER_STORE_MB to CANARDO
REPLICATION& for table MIMER_STORE.MUSIC, MIMER_STORE.BOOKS;
```

Create a subscription for replication of the table MIMER_STORE.MUSIC and MIMER_STORE.BOOKS to the database CANARDO using default value for timing and commit rate. There must exist a schema named MIMER_STORE containing the tables BOOKS and MUSIC on the database CANARDO.

ALTER SUBSCRIPTION

Alters a subscription.



Description

The alter subscription statement is used to add or drop tables to an existing subscription. It can also be used to change the default replication interval or the commit rate.

It is not possible to alter a subscription if the `REPSERVER` program is running for that subscription.

Examples

```
REPLICATION>alter subscription SUB_MIMER_STORE
REPLICATION& drop table MIMER_STORE.MUSIC;
```

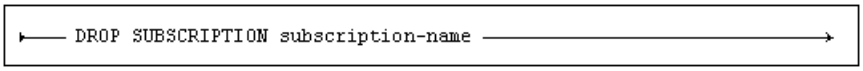
The table MIMER_STORE.MUSIC will not be replicated any more. The triggers defined on this table and the log table will be dropped.

```
REPLICATION>alter subscription SUB_MIMER_STORE commit after 1;
```

Change the transaction rate so that each transaction on the source database will be treated as one transaction on the target database.

DROP SUBSCRIPTION

Drops a subscription.



Description

Drop all information about a subscription from the replication dictionary. It will also drop triggers and the log table, which were created when the subscription was created.

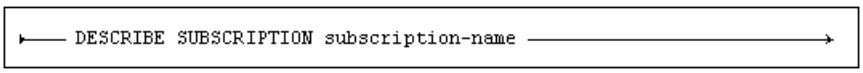
It is not possible to drop a subscription if the `REPSERVER` program is running for that subscription.

Example

```
REPLICATION>DROP SUBSCRIPTION SUB_MIMER_STORE;
```

DESCRIBE SUBSCRIPTION

Describes a subscription.



Description

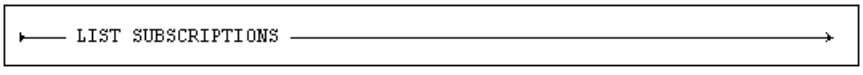
Display information about the specified subscription.

Example

```
REPLICATION>describe subscription SUB_MIMER_STORE_MB;

Subscription SUB_MIMER_STORE_MB:

Target database: CANARDO
Interval:        600 second(s)
Commit after:    1
SYSTEM_STARTUP:  2010-08-27 18:08:52
TRANSNO:         4711
SEQNO:           12
STOPPING_FLAG:   NO

Tables:

Source: MIMER_STORE.MUSIC
Target: MIMER_STORE.MUSIC

Source: MIMER_STORE.BOOKS
Target: MIMER_STORE.BOOKS
```

LIST SUBSCRIPTIONS

Lists subscriptions.



Description

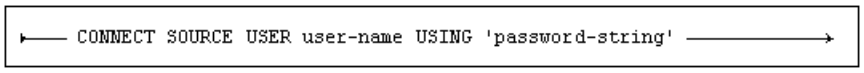
Lists all defined subscriptions

Example

```
REPLICATION>LIST subscriptions;

Subscriptions
=============
MIMER_STORE
MIMER_STORE_MB
```

CONNECT SOURCE USER

Connects user to source database.



Description

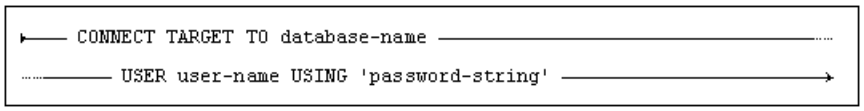
Connect the table owner to the source database.

Example

```
REPLICATION>CONNECT SOURCE USER MIMER_STORE
REPLICATION& using 'GoodiesRUs';
```

CONNECT TARGET USER

Connects user to target database.



Description

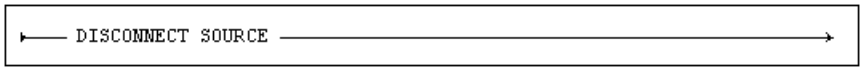
Specify user on target database.

Example

```
REPLICATION>CONNECT TARGET to CANARDO
REPLICATION& user MIMER_STORE using 'niTeoW1';
```

DISCONNECT SOURCE

Disconnects user from source database.



Description

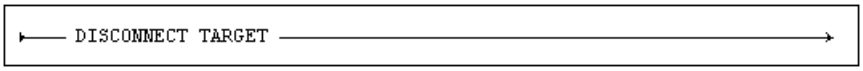
Disconnect the user connected to the source database.

Example

```
REPLICATION>DISCONNECT SOURCE;
```

DISCONNECT TARGET

Disconnects user from target database.



Description

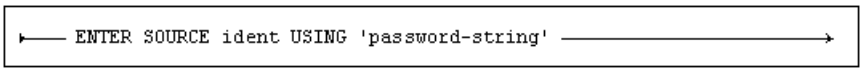
Disconnect the user connected to the target database.

Example

```
REPLICATION>DISCONNECT TARGET;
```

ENTER SOURCE

Connects a PROGRAM ident to source database.



Description

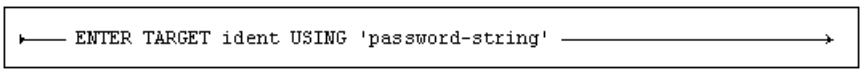
Connects a `PROGRAM` ident to the source database.

Example

```
REPLICATION>ENTER SOURCE 'PgmIdnt' USING 'SecrtPlees';
```

ENTER TARGET

Connects a PROGRAM ident to target database.



Description

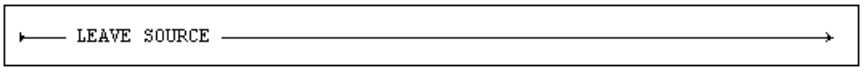
Connects a `PROGRAM` ident to the target database.

Example

```
REPLICATION>ENTER TARGET 'PgmIdnt' USING 'SecrtPlees';
```

LEAVE SOURCE

Leaves a PROGRAM ident from the source database.



Description

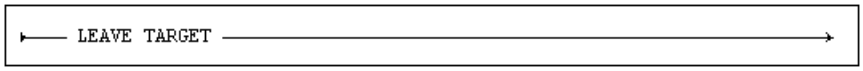
The current source `PROGRAM` ident is left and the saved environment of the previous ident is restored.

Example

```
REPLICATION>LEAVE SOURCE;
```

LEAVE TARGET

Leaves a PROGRAM ident from the target database.



Description

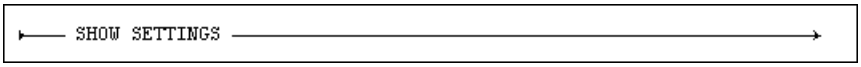
The current target PROGRAM ident is left and the saved environment of the previous ident is restored.

Example

```
REPLICATION>LEAVE TARGET;
```

SHOW SETTINGS

Shows source and target info.



Description

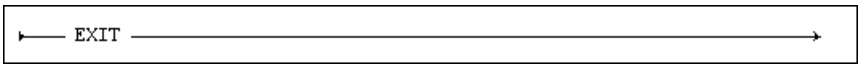
Display information about source and target user.

Example

```
REPLICATION>SHOW SETTINGS;

Settings
========
Source database: MOONBASE_ALPHA
Source user:     MIMER_STORE
Target database: CANARDO
Target user:     MIMER_STORE

Source program:  PgmIdnt
Target program:  Not connected
```

EXIT

Exits MIMREPADM.



Description

Exit from the MIMREPADM program.

Example

```
REPLICATION>EXIT;
```

REPSERVER - Replicating the Data

The actual replication is performed by running the REPSERVER program. This program will handle the replication for one subscription. This program connects to source database as REP_SOURCE_USER and to the target database as REP_TARGET_USER.

MIMSYNC is typically used before replication is first set up, or has been halted for some reason, to make sure that source and target tables have the same contents. After the synchronization, the replication functionality ensures that the tables remain identical.

Syntax

```
repserver [-e|--exit] [-l file|--logfile=file] [-d|--verbose] [--spassword=pass]
    [--spassword=pass] [source_database] [subscription]

repserver -t [--spassword=pass] [source_database] [subscription]

repserver [-v|--version] | [-?|--help]
```

If any required parameter is omitted, the program will prompt for these values.

Command-line Arguments

You can use the following arguments with REPSERVER.

Unix/Windows-style	Function
-d --verbose	More detailed, verbose output.
-e --exit	Exit option.
-l file --logfile=file	Multifile, if omitted standard output.
-t --stop	Terminate replication server.
-v --version	Display version information.
source_database	Specifies the name of the database to access. If specified, it must be the last argument. If you do not specify a database name, the default database will be used.
-? --help	Show help text.
--spassword=pass	REP_SOURCE_USER source password.
--tpassword=pass	REP_TARGET_USER target password.

The REPSERVER program will read the log tables for all tables in the subscription and perform the same operations on the target database. After each commit on the target database the data in the log tables will be deleted. Once all operations have been done the program will sleep for the rest of the interval specified for the subscription. If the interval for the subscription is set to 0 the program will poll the log tables for any data continuously.

Start the Replication

To start the replication for a subscription the REPSERVER program can be started with following command line arguments:

```
repserver [--spassword=password] [--tpassword=password] [--logfile=logfile]
    [--exit] database [subscription-name]
```

Note: The REPSERVER program should normally be run as a detached process on VMS, or as a background process on Linux.

Stop the Replication

To stop the replication the REPSERVER program should be run with the following options

```
REPSERVER –t [--spassword=password] database subscription-name
```

This will set the stopping flag in the replication dictionary to 'YES' for the specified subscription. The REPSERVER program will periodically look at this flag, when not active. This means that it can take some time before the replication is stopped.

Error handling

Most Mimer SQL errors are considered fatal for REPSERVER, with the exception of the following three:

- Error -10101, INSERT operation invalid because the resulting table will contain a primary key duplicate
- Error -10110, unique constraint violation
- Error 100, record for update or delete not found

These errors will only result in a warning. This is to make replication possible even if the target table is not identical to the source table.

If REPSERVER gets a transaction conflict, it will try to execute the transaction once more. If the second attempt fails REPSERVER considers this a fatal error.

MIMSYNC - Synchronizing Tables

A third component in the replication service is the MIMSYNC program. It is typically run in batch and operates on pairs of tables, where one table resides in the source database and the other resides in the target database. Data manipulation operations are performed to ensure that the two tables contain the same rows. The table in the source database is considered to be the master, which means that MIMSYNC will only modify (delete/insert/update) the table in the target database.

Synchronization can in some cases be used instead of replication. If the replication updates only need to be performed, say, every 24 hours, this could be done by a batch job running MIMSYNC each night.

The MIMSYNC program supports synchronization between tables in the source and target database. The program operates on pairs of tables, and compares the contents of the two tables in a pair and makes both contain the same records. The table in the source database is considered to be the master, which means that it is the table in the target database that will be updated. The SQL statements needed to modify the target table are constructed and grouped into reasonably large transactions (1 000 rows).

Syntax

```
mimsync -s|--subscription [-n|--noexecute] [-l file|--logfile=file]
    [subs-options] [source_database] [subscription]

mimsync -t|--table [-n|--noexecute] [-l file|--logfile=file]
    [table-options] [source_database]

mimsync [-v|--version] | [-?|--help]
```

Options

Unix/Windows-style	Function
`-l file` `--logfile=file`	Logfile, if omitted standard output.
`-n` `--noexecute`	Do not update target, verify only.
`-s` `--subscription`	Synchronize a subscription.
`-t` `--table`	Synchronize a table.
`-v` `--version`	Display version information.
`source_database`	Specifies the name of the database to access. If specified, it must be the last argument. If you do not specify a database name, the default database will be used.

Unix/Windows-style	Function
subscription	Subscription name.
-? --help	Show help text.

Subs-options

Unix/Windows-style	Function
--spassword=pass	REP_SOURCE_USER source password.
--tpassword=pass	REP_TARGET_USER target password.

Table-options

Unix/Windows-style	Function
--stable=table	Source table.
--ttable=table	Target table.
--susername=user	Source user.
--spassword=password	Source password.
--sprogram=program	Source program.
--susing=password	Source program password.
--tdatabase=database	Target database.
--tusername=user	Target user.
--tpassword=password	Target password.
--tprogram=program	Target program.
--tusing=password	Target program password.

If the database, user and/or password switches are not given the program will prompt for database, user and/or password. The other switches are optional.

Examples

```
$ mimsync -s --logfile=synclog --spassword=secret --tpassword=secret
    sourcedb subs1

$ mimsync -t --suuser=SrcUsr --spassword=scrt
    --tdatabase=TrgDb --tuser=TrgUsr --tpassword=scrt
    table1 table2 SrcDb
```

Authorization

When synchronizing a subscription MIMSYNC is run as REP_SOURCE_USER and REP_TARGET_USER.

When synchronizing a pair of tables MIMSYNC can be run as any user (having SELECT and INSERT rights).

Restrictions

- The tables must have a primary key
- The tables must have the same definition

Note: MIMSYNC may fail to synchronize a table that has a foreign key reference to itself. Also circular foreign keys may cause problems.

Output

When executing MIMSYNC, it will write execution information to its log file. (If no log file is specified, standard output will be used.)

Example log file:

```
2019-11-29 09:33:41.62   <Information>
------------------------------------------------------------------------
Starting Mimer SQL Synchronization

2019-11-29 09:33:42.74   <Information>
========================================================================
Mimer SQL 11.0.1A
Mimer SQL Synchronization
for subscription S on database SOURCEDB STARTED

Synchronizing table SYSADM.T1...
Synchronization of table SYSADM.T1 complete

Synchronizing table SYSADM.T3...
Synchronization of table SYSADM.T3 complete

2019-11-29 09:33:42.98   <Information>
------------------------------------------------------------------------
Mimer SQL Synchronization for subscription S STOPPED
```

Chapter 10

Mimer SQL Shadowing

Mimer SQL Shadowing makes it possible to create and maintain one or more simultaneously updated copies of a databank. This allows for a higher degree of data availability by giving extra protection from disk crashes, etc.

This chapter describes the functions and benefits of databank shadowing, how to use Mimer SQL Shadowing, and how to handle problems.

Note: Mimer SQL Shadowing is not included in the standard Mimer SQL distribution. In order to implement shadowing, you must have the correct type of license. Contact Mimer Information Technology AB for more information.

About Databank Shadowing

Databank shadowing means updating one or more copies of a databank simultaneously.

The master is the 'normal' databank file which is accessed for data storage and retrieval. The copies are called shadows. A databank can have more than one shadow.

Any changes to the master databank are automatically made to the shadow, thus protecting data from a disk crash or other event that might cause a databank to be lost.

If a master databank is lost, a shadow will automatically take over from the master and operations can be resumed immediately, assuming the shadow is not also damaged.

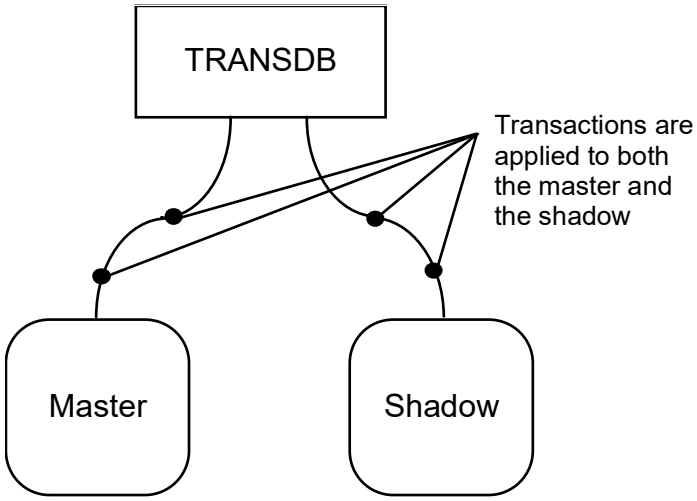
A shadow can be transformed to a master databank to permanently replace it and this process is much faster than restoring a databank from a backup copy.

Using offline shadows provides a straightforward way of backing-up your databanks. Once you have completed the backup and set the shadow online, all operations performed on the master databank are applied to the shadow automatically.

Databank shadowing is entirely invisible to an application. This means that shadows can be added to existing applications. No special handling is needed to access tables in a shadowed databank.

Shadowing Requirements

A databank must support transaction control, see *Transaction Control* on page *17*, to be shadowed.



Databanks with the `WORK` option cannot be shadowed, because shadowing requires transaction handling.

The databank to be shadowed cannot be used by any other users while a shadow is being created.

The shadow name cannot be the same as the name of the master databank, of any other shadow, or of any shadow that has been transformed to a master.

SYSDB and Shadowing

Because the `SYSDB` databank holds all the data dictionary information about your database, protecting it with shadowing and/or backups is essential.

Otherwise, if `SYSDB` is lost, the whole database will be unreachable.

SQLDB and Shadowing

Shadowing `SQLDB` is not allowed, as it is a `TEMPORARY` databank, and not needed because `SQLDB` only contains temporary data.

Creating Shadows

When you create a shadow for a databank, all tables and indexes in the databank are copied to the shadow.

Creating a shadow for a large databank may take some time, and thus should be carefully planned.

Altering Shadows

When you alter a shadow to a master, it only affects the Mimer SQL data dictionary. The databank filenames are not changed. The databank cannot be used by any other user when a shadow is being transformed into the master (this is not very likely to happen since this function is normally used when the master has been lost or damaged).

Backups

We recommend that you take conventional backups as a supplement to databank shadows to protect data in the event of a crash that destroys both the master and the shadow.

Because operations are not interrupted when shadow-backups are taken, and because Mimer SQL databanks are automatically reorganized, you get true 24 hour-a-day operation.

Dropping Shadows

When a shadow is dropped, the file where the shadow is stored is usually deleted from the file system. If it is not deleted, you can use operating system facilities to delete it.

The databank cannot be used by any other users while a shadow is being dropped.

Levels of Data Protection

Backing-up and Restoring Data on page *67*, describes the role of the system databanks `LOGDB` and `TRANSDB` when used in conjunction with backup and restore, in protecting data against loss.

Databank shadowing provides an even higher level of protection. Listed below are the different ways in which data can be protected from loss (from the least amount of protection to the highest).

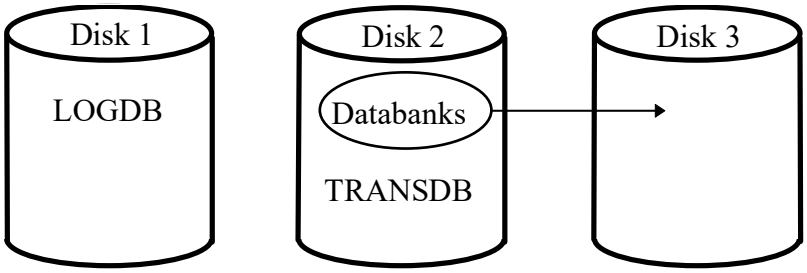
All Databanks on One Disk and No Logging

If a databank is lost with this level of protection, it is only possible to continue operations from the last backup copy (all changes since the last backup was taken are lost).

Databanks can be lost due to accidental deletion, disk crashes (which can destroy all files on a disk), etc. This level of protection is not recommended except for trash databanks with unimportant contents.

Logging, with LOGDB and TRANSDB on a Separate Disk from the Data

LOGDB and TRANSDB are vital databanks if the system stops or if any databanks are lost. Because of this, LOGDB and TRANSDB should be stored on separate disks, as shown in the following figure:



Application data should be stored on the TRANSDB disk if it cannot be stored separately.

If a databank is lost, it can be restored to its original state by applying the transactions in LOGDB and TRANSDB to a restored backup of the databank.

This may take some time, especially if the databank is large and if there is a lot of transaction information stored in LOGDB.

Caution:	If the databank disk and the TRANSDB or LOGDB disk are handled by the same disk controller, a disk controller failure may cause both disks to crash. If this happens, the databanks can only be restarted from the state of the last backup copy. Therefore, we advise you to use separate disks with separate disk controllers.

This security level gives a high degree of security and is recommended for databanks containing important data used in a system where the delay before the system is restored after a crash is not critical.

To assure this high degree of security, backup files should always be stored on separate removable media (e.g. CD/RW).

Shadowing, with Shadows on a Separate Disk

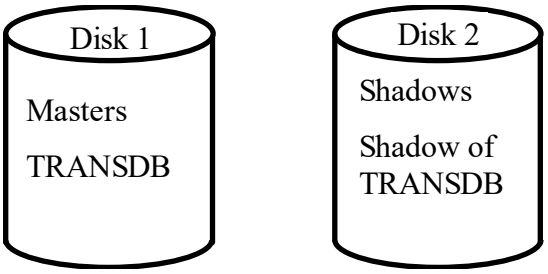
Shadows should always be stored on a separate disk from the masters to protect them from a total disk crash that could destroy both the master and shadow databanks.

It is also advisable to use separate disk controllers to assure that a corrupt disk controller does not destroy the disks holding both the masters and the shadows.

If a databank is lost, its shadow can be transformed into a master and the shadow automatically takes over with no loss of data.

Since shadows are updated after the master, and operations are saved in TRANSDB until the shadow is updated, it is important that TRANSDB is consistent when a shadow is transformed.

To ensure this, you should shadow TRANSDB. We strongly recommend that TRANSDB and its shadow are stored on separate disks, as shown in the following diagram:



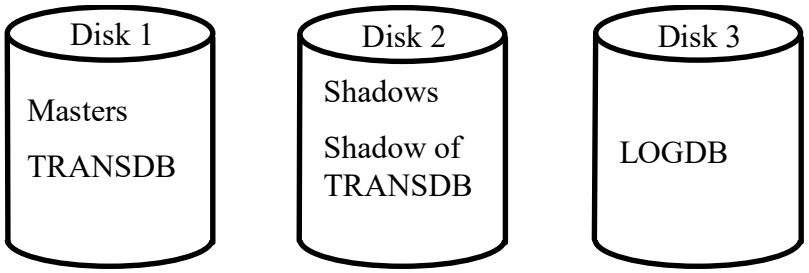
This arrangement gives a high degree of security and is recommended for databanks containing important data used in a system where it is vital to be able to get the system running again quickly after a disk crash.

Shadowing and Logging

Combining shadowing and logging, see *Backing-up and Restoring Data* on page *67*, gives the highest level of data protection.

If logging is used, the data is protected if both the master and the shadow disks become corrupted.

And, when shadowing is combined with logging (with LOGDB on a third disk) and backups are regularly taken to separate media (CD/RW, etc.), then data is always protected if any two of the disks crash, for example:



Of course, additional disks can be used, just as long as the databanks that are separated above are not placed on the same disks. If you only have two disk drives available and all the databanks are shadowed, then logging is of little value. Shadowing LOGDB will not increase data protection significantly in this configuration.

Different degrees of data security can be used for different databanks, depending on the importance of the data. It is however important that all inter-dependent databanks (because of foreign key relationships, etc.) have the same level of protection. Otherwise logical inconsistencies may result if there is a disk crash.

Creating and Managing Shadows

Shadows are managed using the SQL shadowing commands:

`CREATE SHADOW` – creates a shadow

`SET SHADOW` – sets shadows on- or offline

`ALTER SHADOW` – swaps a shadow with its master

`DROP SHADOW` – drops a shadow.

Note: For shadow information, use the `LIST SHADOWS` command in BSQL.

Privileges

If you have `SHADOW` or `BACKUP` privilege, you can list shadowing information for all databanks.

Users can always backup and restore, set shadows offline and online, and list shadowing information for any databank that they have created.

`SYSADM` is initially granted `SHADOW` and `BACKUP` privilege with the `WITH GRANT OPTION`.

The following table shows the privileges you need to work with shadows:

Tasks	**Privilege**	
	Backup	**Shadow**
Create a shadow *)		X
Drop a shadow *)		X
Transform a shadow *)		X
List shadow info	X	X
Backup and restore	X	
Set shadow offline	X	

*) You must have exclusive use of the databank. This means that no other user can access the databank until the operation is finished.

SQL Shadowing Commands – an Example Session

In the following sections, we use an example session to show how to:

- create shadows
- set them offline and take backup
- set them online again
- restore both user databanks and shadows
- drop shadows.

About the Examples

The examples in the following sections are based on:

Ident:	BACADM
Ident Privileges:	DATABANK, SHADOW and BACKUP privileges
User databanks:	ARTICLES, CUSTOMERS with the LOG option enabled
System databanks:	TRANSDB, SYSDB and LOGDB

We assume that you are the ident BACADM and have connected to the database using Mimer BSQL, as follows:

```
SQL> CONNECT;
User: BACADM
Password: Masterp1ece
```

Creating a Shadow

The following example creates shadows for the user and system databanks:

```
SQL> CREATE SHADOW   TRANSDB_SH   FOR TRANSDB   IN 'transdb_sh.dbf';
SQL> CREATE SHADOW   SYSDB_SH     FOR SYSDB     IN 'sysdb_sh.dbf';
SQL> CREATE SHADOW   LOGDB_SH     FOR LOGDB     IN 'logdb_sh.dbf';
SQL> CREATE SHADOW   ARTICLES_SH  FOR ARTICLES  IN 'articles_sh.dbf';
SQL> CREATE SHADOW   CUSTOMERS_SH FOR CUSTOMERS IN 'customers_sh.dbf';
```

For information on the CREATE SHADOW command, see the *Mimer SQL Reference Manual, Chapter 12, CREATE SHADOW*.

Setting a Shadow Offline

You must set a shadow offline, for example, when backing-up a databank, to ensure that the databank shadow file is in a consistent state when the backup copy is taken.

The following example sets all the shadows created in the previous example offline:

```
SQL> SET SHADOW TRANSDB_SH,LOGDB_SH,SYSDB_SH,ARTICLES_SH,CUSTOMERS_SH
OFFLINE;
```

For information on the SET SHADOW command, see the *Mimer SQL Reference Manual, Chapter 12, SET SHADOW*.

Caution: TRANSDB stores all operations carried out while shadows are offline. We recommend that you always set shadows online as soon as possible. If you do not, you risk TRANSDB filling disk capacity.

Backing-up from Shadows

You can back-up a databank using its shadow instead of the master databank.

This allows the backup process to proceed without affecting the users working with data contained in the databank.

When a shadow is set offline, the relevant transactions will be written to the online databank and remain in TRANSDB until the shadow is set online again and the transactions can be written to it.

To back-up your database from shadows:

1. Set the shadows offline, as shown in the previous example.
2. Use your operating system's functionality to copy the shadow files. For example, on Linux:

```
$ cp sysdb_sh.dbf      sysdb_sh.bac
$ cp transdb_sh.dbf    transdb_sh.bac
$ cp logdb_sh.dbf      logdb_sh.bac
$ cp articles_sh.dbf   articles_sh.bac
$ cp customers_sh.dbf  customers_sh.bac
```

3. Set the shadows online and reset the log, as shown in the next example.
4. Move the backup files to a safe medium, such as CD/RW. Your backup is complete.

Setting a Shadow Online

When you set databank shadows online, they are automatically updated in the background to the current state of the master database.

Continuing with the previous example, you can set the shadows online, as follows:

```
SQL> SET SHADOW SYSDB_SH,LOGDB_SH,TRANSDB_SH,ARTICLES_SH,CUSTOMERS_SH ONLINE
RESET LOG;
```

Note: To get a backup timestamp, to be able to use the log when restoring from this backup, the `RESET LOG` option is used.

For information on the `SET SHADOW` command, see the *Mimer SQL Reference Manual, Chapter 12, SET SHADOW*.

Restoring a User Databank

If an error is encountered on a user databank, the system will continue to operate using the databank shadow. You can restore the damaged user databank by swapping it with its shadow using the `ALTER SHADOW` command.

For example, if the `ARTICLES` databank has been damaged, you can restore it by swapping it with its shadow `ARTICLES_SH` using the `ALTER SHADOW` command:

```
SQL> ALTER SHADOW ARTICLES_SH TO MASTER;
```

Now, the faulty `ARTICLES` databank is the shadow. However, it has the same name as the master databank.

To return to our original situation, we must delete (drop) the faulty shadow, create a new shadow and swap the shadow with the master so that the databanks are correctly named.

```
SQL> DROP SHADOW ARTICLES;
SQL> CREATE SHADOW ARTICLES FOR ARTICLES_SH IN 'articles.dbf';
SQL> ALTER SHADOW ARTICLES TO MASTER;
```

The first command deletes the shadow.

The second command creates a shadow for the master with the original name and location.

The third command swaps the shadow with the master.

Note: If the original situation is not restored as shown above, the shadow name, in this case ARTICLES_SH, will remain allocated internally which could be confusing.

For information on the ALTER SHADOW command, see the *Mimer SQL Reference Manual, Chapter 12, ALTER SHADOW*.

Restoring Both a User Databank and Its Shadow

If both a user databank and its shadow are lost or damaged, you can restore the data using the shadow's backup files and LOGDB.

1. We recommend that you stop the Mimer SQL database server when replacing databanks.
2. Copy the shadow's backup file to the position of the damaged databank file, for example, on a Linux system:

```
$ cp articles_sh.bac articles.dbf
```

3. Restart the Mimer SQL database server and use the ALTER DATABANK RESTORE command to restore the databank:

```
SQL> ALTER DATABANK ARTICLES RESTORE USING LOG;
```

Note: The restore command above will automatically recreate the corresponding shadow databank.

For more information on ALTER DATABANK RESTORE, see the *Mimer SQL Reference Manual, Chapter 12, ALTER DATABANK RESTORE*

Restoring System Databanks

You cannot use the ALTER SHADOW command to swap the system databanks SYSDB, TRANSDB and LOGDB with their shadows in order to restore them. You must alter them using the BSQL program.

For more information, see:

- *Transforming a SYSDB Shadow to a Master* on page *132*
- *Transforming a TRANSDB Shadow to a Master* on page *133*
- *Transforming a LOGDB Shadow to a Master* on page *133*.

Dropping a Shadow

As seen in a previous example, you can delete a shadow by dropping it, for example:

```
SQL> DROP SHADOW ARTICLES_SH;
```

For information on the DROP SHADOW command, see the *Mimer SQL Reference Manual, Chapter 12, DROP*.

Shadowing System Databanks

The system databanks (SYSDB, TRANSDB, LOGDB and SQLDB) require special handling in some situations.

If a problem occurs with these databanks or their shadows, the only permitted login is SYSADM logging into the BSQL program. The BSQL program will then recognize the problem and help you correct it.

System databanks are handled in the same way as other databanks, with the following exceptions:

- If an error is encountered on a user databank, automatic shadowing fail-over takes place. However, if there is a problem with SYSDB, TRANSDB, SQLDB or LOGDB, new users cannot login. Users already active will receive an error message when attempting operations that depend on the affected system databank, while other operations continue to work. The error state is held until Mimer SQL is stopped and the error is corrected.
- If there is a problem with SYSDB, TRANSDB or LOGDB shadows, new users cannot login until the faulty shadow is dropped or suspended, see *If a Shadow for SYSDB, TRANSDB or LOGDB Is Not Accessible* on page *134*.
- No users can be connected while a shadow for SYSDB, TRANSDB, or LOGDB is being created, altered or dropped.
- You cannot create a shadow for SQLDB as it is a TEMPORARY databank.

Transforming a SYSDB Shadow to a Master

If SYSDB is lost or corrupt, any existing SYSDB shadow can be altered to become the master in order to allow Mimer SQL to start again.

The SYSDB shadow file should be renamed and/or moved to the location where the master SYSDB was. Then the BSQL is started and login is performed as SYSADM. Enter the name of the shadow to be transformed into the master, and exit.

Example

```
Mimer SQL command line utility, version 11.0.7A
Username: SYSADM
Password:
MIMER/DB warning -18013 in function CONNECT
         MIMER/DB started from SYSDB shadow. Transform SYSDB shadow to master
         with BSQL, or restart system from master SYSDB

 -- Transform shadow --

DATABANK
SHADOW
OFFLINE
FILE
=======================================
SYSDB
SYSSH
N
SYSDB_S
---
One shadow found
Name of shadow to transform (<CR> = skip): sysdb_s

Shadow SYSSH transformed to master
```

If the disk where SYSDB is located becomes inaccessible, it may be more suitable to redefine the database home directory (to point out the SYSDB shadow) instead of restoring the original directory structure.

Note: In this case the ALTER DATABANK statement must be used for all databanks explicitly defined to be located on the halted disk, i.e. with an absolute file specification in the data dictionary.

Restoring SYSDB

If SYSDB is lost and no shadows exist, a backup copy of SYSDB can be restored to allow Mimer SQL to start again, an example of how to do this is given in *Backing-up and Restoring Data* on page *67*.

TRANSDB and Shadowing

Shadowing TRANSDB assures that you can bring your database up-to-date if the TRANSDB master is lost or damaged.

Transforming a TRANSDB Shadow to a Master

Start the BSQL program and login as SYSADM. A message is displayed saying that TRANSDB cannot be opened, and a shadow must be transformed to the master, this is similar to the example in *Transforming a SYSDB Shadow to a Master* on page *132*.

If there are uncompleted transactions they will be completed, as if the original TRANSDB was still functioning.

LOGDB and Shadowing

If some databanks are not shadowed but backup copies of the databanks exist, then a shadow of LOGDB is useful since LOGDB, in this case, is even more important from a restore perspective.

Transforming a LOGDB Shadow to a Master

Start the BSQL program and login as SYSADM. A message is displayed saying that LOGDB cannot be opened, and a shadow must be transformed to the master, this is similar to the example in *Transforming a SYSDB Shadow to a Master* on page *132*.

If there are transactions not yet written to the log, they will be written automatically.

SQLDB and Shadowing

Shadowing SQLDB is not necessary because SQLDB only contains temporary data.

However, SQLDB is required when a user logs on to Mimer SQL. If SQLDB is corrupt or lost, you must recreate it by logging on to the BSQL program as SYSADM. This automatically recreates SQLDB if the databank is not found.

If a Shadow for SYSDB, TRANSDB or LOGDB Is Not Accessible

If a shadow for SYSDB, TRANSDB or LOGDB is not accessible, SYSADM should login to the BSQL program.

An error message is given followed by the option to drop the shadow or set it offline. If the shadow is corrupt or missing, you should drop it. For example:

```
Mimer SQL command line utility, version 11.0.7A
Username: SYSADM
Password:
MIMER/DB fatal error -16142 in function CONNECT
         Cannot open databank LOGDB_S,
         file logdb_s not found

Inaccessible shadow encountered. DROP or SET OFFLINE? (D/S): D

                                -- Drop shadow --

Shadow LOGDB_S dropped
```

If the shadow is only temporarily unavailable, it may be enough to set it offline for a short period of time.

Data Protection Strategy

So far, this chapter has described the facilities that are available for a system manager to maintain the system.

But how do you know that you are using the functionality in the right way, and that you will be able to get the system running again if something happens?

The answer is planning and practice. When you have planned a data protection strategy, you should put it into practice by simulating a disk crash and restoring the databanks according to your strategy.

Check that the contents of all tables are the same as before the simulated crash. Do not forget to simulate a crash of the system databanks (SYSDB, TRANSDB, LOGDB and SQLDB) and then restore or recreate them.

Once you have a successful strategy, build command files (scripts) that perform the correct operations regularly.

Configuring Your System

In *Levels of Data Protection* on page *125*, we discussed various levels of data security.

When you configure your system, there are some additional questions to be answered:

- **How do I divide the workload over several disks to get the best performance?**

  TRANSDB and its shadows should preferably be on fast disks, see *Performance Aspects of Shadowing* on page *135*.

- **How much disk space do I need for shadowing?**

  A shadow file occupies the same amount of disk space as the master databank file.

  In general, you will have problems if TRANSDB or LOGDB run out of disk space. When this happens the system cannot continue. Therefore, it is important to make sure that there is enough space for these databanks.

  How large TRANSDB grows depends on the number of transactions and how fast the background threads are able to perform the transactions on the shadows. When the transactions are performed on the shadows, the space in TRANSDB where they were stored is released.

  How large LOGDB grows depends on whether the LOG option is used and how often backups are taken and DROP LOG is performed, since DROP LOG clears LOGDB and releases space (but the file size remain unchanged).

  Note: Try and keep the length of time the shadows are offline to a minimum. If you do not, you risk TRANSDB growing and filling the disk to capacity.

Performance Aspects of Shadowing

In multi-user systems, performance is not noticeably affected by shadowing, even though the shadowing system needs more machine resources because more files need to be updated.

Applications do not have to wait for the shadows to be updated as this is done in the background. Actually, all updates to the disk, even to master databanks, are performed by background processes (except for updates to TRANSDB and its shadows).

In single-user systems, no background process is used. This means that an application has to wait for the shadows to be updated.

Troubleshooting

If shadow updating is delayed TRANSDB grows. This can happen for several reasons:

- A shadow has been set offline and forgotten. If this happens, transactions will be buffered until the shadow is set online again.

  To check if a shadow is offline, in BSQL use the LIST SHADOWS command or the *The Performance Report* on page *59*.

- A shadow is corrupt. Updates to the shadow result in an I/O error, and are buffered in TRANSDB.

  When this happens, the operator is notified by the system. To check, for notification, use the *The Performance Report* on page *59* and check your Mimer SQL database server log, see *Database Server Log* on page *65*.

- There are too few background threads to update the shadows, or they get too little machine resources.

Chapter 11

Database Statistics

The SQL statistics statements collect statistical information about table and index data in the database and store this information in the data dictionary.

The information is used by the SQL compiler in optimizing access paths for SQL queries.

The statistical information includes:

- the total number of rows in each base table
- the number of distinct values in each column of a table
- the number of non-null values in each column of a table
- the lowest and highest values in each column of a table

Authorization

The user executing the SQL statistics statements must either have `STATISTICS` privilege or be the owner of the table(s) or ident(s) for which statistics are being collected.

The database administration ident `SYSADM` holds `STATISTICS` privilege with the `WITH GRANT OPTION`, and may thus take responsibility for maintaining statistics for the whole system or delegate the responsibility to selected idents.

Note: A user with `STATISTICS` privilege is not necessarily permitted to read the contents of the databank using data manipulation statements, this privilege only permits access for the collection of statistics.

The SQL Statistics Statements

The SQL statistics statements `UPDATE STATISTICS` may be used to collect statistical information in the areas described below. Also refer to the *Mimer SQL Reference Manual, Chapter 12, UPDATE STATISTICS* for details.

Statistics may be collected for the entire database, i.e. all tables in all databanks recorded in the same `SYSDB`, for tables owned by specified idents, or for specific tables.

The statement `DELETE STATISTICS` is used to remove the statistics collected. See *Mimer SQL Reference Manual, Chapter 12, DELETE STATISTICS* for details.

Note: The database remains fully accessible while statistics are being collected (or deleted).

Statistics for the Entire Database

To collect statistical data for all tables in the database, use the following function:

```
SQL> UPDATE STATISTICS;
```

The user must have STATISTICS privilege.

Note: Even in a database of only moderate size, collecting statistical data for all tables is time-consuming. We recommend that you run this option in particular at off-peak times.

Statistics for Specified Idents

To collect statistics for all base tables belonging to schemas owned by a list of specified idents, use the following function:

```
SQL> UPDATE STATISTICS FOR IDENT list-of-idents;
```

A user requesting statistics for tables belonging to a schema owned by an ident other than himself must have STATISTICS privilege.

To collect statistics for SYSDB, the pseudo-ident SYSTEM may be specified.

Statistics for Specified Tables

To collect statistics for a list of specified tables, use the following function:

```
SQL> UPDATE STATISTICS FOR TABLE list-of-tables;
```

The user requesting statistics for the tables specified in the list must either be the owner of them or have STATISTICS privilege.

Secondary Index Consistency

The update statistics facility includes an automatic function which ensures that all secondary indexes on tables contained in databanks with the TRANSACTION or LOG option are in a consistent state.

This function is performed in a way that makes it transparent to other users of the database and it is only performed on secondary indexes created on tables actually selected by the UPDATE STATISTICS statement.

It will take some time to verify the consistency of a secondary index. The data dictionary table TABLE_CONSTRAINTS can be used to determine which secondary indexes are flagged as not consistent (shown in the column named IS_CONSISTENT).

An index which is in a consistent state will offer optimal performance when used in a query.

All secondary indexes contained in a databank with the WORK option and those contained in a databank that has been upgraded from Mimer SQL version 7 or 8.1 will be flagged as not consistent.

When to Use the SQL Statistics Statements

Mimer SQL collects basic statistics for each table whenever the table is opened. These statistics may suffice for maintaining high performance in many situations.

If optimal performance is required for an application, the SQL statistics statements should be used to collect detailed information (this includes information on value distribution and table size).

When this is the case, statistics should be typically updated in the following situations:

- when the size of a table has changed significantly
- when the maximum/minimum limits on values in a table have altered significantly
- when a databank has been altered from having the `WORK` option to having the `TRANSACTION` or `LOG` option and contains secondary indexes
- when a databank with the `TRANSACTION` or `LOG` option contains secondary indexes and has just been upgraded from Mimer SQL version 7 or 8.1.

The statistics information in the data dictionary is used only by the Mimer SQL compiler.

Note: Only the performance, not the result, of an SQL statement is affected by gathering and using the statistical information and by ensuring the consistency of secondary indexes.

Chapter 12

SQL Monitoring on the Database Server

The SQL monitor program provides functionality for monitoring SQL statement usage on a Mimer SQL database server. The tool can be used to locate expensive SQL statements, or to understand which application is using what resources in the database server.

SQLMONITOR - SQL Monitoring

Syntax

`SQLMONITOR` is controlled by the following command-line parameters:

```
sqlmonitor [-u user] [-p pass] [-e prog] [-i pass] [-n secs]
[-s loops] [-l id] [-d level] [-o column] [-t rows] [-b]
[-w mode] [database]

sqlmonitor [--username=user] [--password=pass]
[--program=prog] [--using=pass] [--interval=secs]
[--stop=loops] [--sqlid=id] [--detail=level] [--order=column]
[--top=rows] [--benchmark] [--wrap=mode] [database]

sqlmonitor [-v|--version] | [-?|--help]
```

Command-line Arguments

Unix-style	VMS-style	Function
`-b` `--benchmark`	`/BENCHMARK`	Control two different snapshots and compare them. Not compatible with the interval and stop switches.

`-d` *level* `--detail=`*level*	`/DETAIL=`*level*	Detail level of output. Valid options are 1, 2 or 3. If omitted, defaults to 1, unless at least one sqlid switch is given, in which case it defaults to 2. 1 = minimal amount of information, statements excluded. 2 = all numerical information, statements excluded. 3 = all information, statements included.
`-e` *program* `--program=`*program*	`/PROGRAM=`*program*	Name of a program to enter and show statistics for. If both this and the program password switches are omitted, no program will be entered and statistics will be shown for the originally given ident. If one of the two are given, the other one is prompted for.
`-i` *password* `--using=`*password*	`/USING=`*password*	Password of a program to enter and show statistics for.
`-l` *id* `--sqlid=`*id*	`/SQLID=`*id*	ID of one or more specific SQL statements(s) to show (integer > 0). Multiple switches show multiple statements. Will show all statements if omitted. Will only show a statement if the given ident is permitted to view that specific statement.
`-n` *seconds* `--interval=`*seconds*	`/INTERVAL=`*seconds*	The interval with which to monitor the database, in seconds (integer > 0). If omitted, a single snapshot will be taken.

`-o` *column*
`--order=`*column*
	`/ORDER=`*column*	What column to order the result by. If omitted, table_ops will be used.
		Valid options are:
		`table_ops`
		`table_ops_per_sec`
		`prepare_count`
		`execute_count`
		`server_requests`
		`transaction_record_count`
		`elapsed_time`
		`sql_id`
		`sql_statement`

`-p` *password* `--password=`*password*	`/PASSWORD=`*password*	Password for ident. If the switch is omitted the user is prompted for a password, unless OS_USER is specified as described above.
`-s` *loops* `--stop=`*loops*	`/STOP=`*loops*	After how many intervals to stop monitoring (integer > 0). If the interval switch is given but not the stop switch, monitoring will continue infinitely. If the stop switch is given but not the interval switch, interval will default to 10 seconds.
`-t` *rows* `--top=`*rows*	`/TOP=`*rows*	Use only the top x entries of the result set (integer > 0). Sorting occurs first.
`-u` *username* `--username=`*username*	`/USERNAME=`*username*	Ident name to be used when connecting to database server. If the switch is not given the user is prompted for a username. To connect using OS_USER, give `-u ""`, `--username=""` or `/USERNAME=""`, or leave the username empty when prompted.
`-v` `--version`	`/VERSION`	Display version information.

`-w` *mode*
`--wrap=`*mode*
	`/WRAP=`*mode*	If console output should be truncated, wrapped or neither. If omitted, the text will by default not be altered.
		Valid options are:
		`wrap`
		`none`
		`truncate`

`-?` `--help`	`/HELP`	Display usage information and exit.
`[database]`	`[database]`	Target database name. If omitted, the environment variable MIMER_DATABASE is used if defined, else the default database in SQLHOSTS is used.

Columns

Table operations

The number of operations that have read, inserted, updated or deleted a row in a table when running the given SQL statement.

Note that when a secondary index is used there is one operation to retrieve the data from the index table and one operation from the actual base table. When index lookup only is used by the SQL optimizer the base table is not accessed.

Prepare count

The number of times the given SQL statement has been prepared for execution. This is the number of times the SQL statement has been sent to the server to compile. If a statement has been compiled previously the server will reuse that compilation.

Execute count

The number of times the given SQL statement has been executed. Each select statement will increase this counter once per result set.

Server requests

The number of requests to the server that have been sent in order to run the given SQL statement.

Transaction record count

The transaction overhead caused by running the given SQL statement. The count is the number of rows written to the read and write set during transaction build-up. Note that read-only transaction do not need to write any rows as the system automatically provides a consistent view of the database.

Elapsed time

The amount of time (in seconds) the given SQL statement has spent on the server. Elapsed time is only aggregated if the server has the "timing" setting set to "on". This setting is off by default, but running SQLMONITOR will turn it on.

SQL ID

A serial number that the server appoints an SQL statement when it is first compiled on the server. The ID will remain until the server closes the statement when it is removed from the server's cache of compiled statements.

Multiple seemingly identical SQL statements can appear with different SQL IDs, if they are run by different users on different tables, as they do not represent the same server action.

SQL Statement

The actual SQL statement that has been run.

In the following table the effects of a `SELECT` that are re-opened are modelled:

	Operation	Counter	Comment
1	`PREPARE` (compile) `SELECT`	1 server communication 1 prepare count	
2	`EXECUTE`/`OPEN`		Nothing happens here as the operation is cached until the first fetch.
3	First `FETCH`	1 server communication 1 execute Table operations	
4	Subsequent `FETCH`	Nothing or 1 server communication and table operations	For example, after 200 `FETCH` there is one more server communication and more table operations.
5	`CLOSE` statement/cursor		This operation is typically cached. In some circumstances there is a server communication.
6	New `EXECUTE`/`OPEN`		
7	First `FETCH`	Same as 3	
8	Subsequent `FETCH`	Same as 4	
9	...		

Examples

The parameter options can be combined in the following ways. Each example below is given in both VMS-style and Unix-style.

- Take and print a snapshot of the table operations and elapsed time history of all the SQL statements that have been run by any ident on the database db_name since it was started and are still in use or in the server's cache:

```
SQLMONITOR /USERNAME=SYSADM /PASSWORD=sysadm_password db_name

sqlmonitor -u SYSADM -p sysadm_password db_name

sqlmonitor --username=SYSADM --password=sysadm_password db_name
```

- Take and print a snapshot of the table operations and elapsed time history of all the SQL statements that have been run by the program ExampleProgram on the default database since the database server was started. SYSADM must have execute privilege on ExampleProgram:

```
SQLMONITOR /USERNAME=SYSADM /PASSWORD=sysadm_password
/PROGRAM=ExampleProgram /USING=program_password

sqlmonitor -u SYSADM -p sysadm_password -e ExampleProgram
-i program_password

sqlmonitor --username=SYSADM --password=sysadm_password
--program=ExampleProgram --using=program_password
```

- Take and print a snapshot with all details of the top 10 most expensive SQL statements, based on the number of server requests, that have been compiled on the default database by ident ExampleUser since the database server was started:

```
SQLMONITOR /USERNAME=ExampleUser /PASSWORD=example_password
/ORDER=server_requests /TOP=10 /DETAIL=3

sqlmonitor -u ExampleUser -p example_password -o server_requests -t 10 -d 3

sqlmonitor --username=ExampleUser --password=example_password
--order=server_requests --top=10 --detail=3
```

- Monitor and print numerical detail information of the SQL statements run by the ident ExampleUser, taking a snapshot every 30 seconds and comparing it to the previous snapshot, for a total of 1 hour (120 intervals). Truncate the console output when it reaches the console bounds:

```
SQLMONITOR /USERNAME=ExampleUser /PASSWORD=example_password /INTERVAL=30
/STOP=120 /DETAIL=2 /WRAP=truncate

sqlmonitor -u ExampleUser -p example_password -n 30 -s 120 -d 2 -w truncate

sqlmonitor --username=ExampleUser --password=example_password --interval=30
--stop=120 --detail=2 --wrap=truncate
```

- Take a snapshot of all the numerical information on the SQL statements with ID 32 and 54 when prompted. Then take another such snapshot when prompted and compare it to the first one, to monitor activity on the two specific statements between the first and second points in time. Wrap the console output when it reaches the console bounds:

```
SQLMONITOR /USERNAME=SYSADM /PASSWORD=sysadm_password /BENCHMARK /SQLID=32
/SQLID=54 /WRAP=wrap

sqlmonitor -u SYSADM -p sysadm_password -b -i 32 -i 54 -w wrap

sqlmonitor --username=SYSADM --password=sysadm_password --benchmark
--sqlid=32 --sqlid=54 --wrap=wrap
```

Authorization

The SYSADM ident will always see all SQL statements that are run by any ident, while other idents will only see the SQL statements that they have run themselves, to prevent unauthorized access to possibly sensitive information in the compiled SQL statements.

Chapter 13

DbAnalyzer - index analysis

Mimer `dbanalyzer` is a tool to provide optimization recommendations for a database schema, based on the analysis of primary keys, unique constraints and indexes. The current version provides the following optimization recommendations:

- Remove unnecessary unique indexes that are duplicate with other unique indexes or primary keys.
- Remove unnecessary unique constraints that are duplicate with any unique indexes, other unique constraints or primary keys.
- Remove unnecessary indexes that are duplicate with other explicitly defined indexes and unique indexes, as well as indexes implicitly defined by unique constraints and foreign keys.
- Suggest primary keys for tables without explicitly defined primary keys, based on the defined unique constraints and unique indexes.

The tool is also used to report the usage statistics of indexes and tables. For each index and table, the tool summarizes how many SQL and stored procedure statements have used the index and accessed the table, since the latest start of the server.

`dbanalyzer` detects and reports found column default value inconsistencies that might cause problems in the future. For example a `SMALLINT` sequence used as default value for an `INTEGER` primary key column stops further inserts after 32767 rows, although the table is far from full according to the primary key. And `CURRENT_USER` as default for a `CHARACTER` column can work fine for years, until a person who has a name with Unicode characters starts using the system.

Command syntax

`dbanalyzer` is controlled by the following command-line parameters:

```
dbanalyzer [-m|-s] [-u user] [-p password] [-e program] [-i programpass]
            [-c schema] [-t table] [-a] [-g] [database]

dbanalyzer [-v version]
```

Command-line Arguments (Unix):

Argument	Description
`-u username` `--username=username`	Ident name to be used when connecting to database server. If the switch is not given the user is prompted for a username. To connect using OS_USER, give `-u ""`, `--username=""`, or leave the username empty when prompted.
`-m`	Run `dbanalyzer` in multi user mode.
`-s`	Run `dbanalyzer` in single user mode.
`-p password` `--password=password`	Password for ident. If the switch is omitted the user is prompted for a password, unless OS_USER is specified as described above.
`-e program` `--program=program`	Name of a program to enter.
`-i programpass` `--using=programpass`	Password of program to enter.
`-c schema` `--schema=schema`	Name of the schema to be analyzed. If omitted, the username is used as default schema name.
`-t table` `--table=table`	Name of the table to be analyzed. If omitted, all tables of the analyzed schema will be analyzed.
`-g` `--statistics`	Report the usage statistics of tables and indexes.
`-a` `--analysis`	Report the schema analysis of tables. If neither `-a` nor `-g` is specified, schema analysis is performed by default.
`-v` `--version`	Print version information.
`[database]`	Specifies the name of the database. If a database is not specified, the default database will be monitored. The default database is determined by the setting of the MIMER_DATABASE environment variable.

Command examples

The following command analyzes all tables and their indexes in database DBNAME that user USRNAME with a password USRPASS has access to:

```
dbanalyzer --username=USRNAME --password=USRPASS DBNAME
```

The following command reports usage statistics of all tables and their indexes in database DBNAME that user USRNAME with a password USRPASS has access to:

```
dbanalyzer --username=USRNAME --password=USRPASS --statistics DBNAME
```

To include both schema analysis and usage statistics of the above tables and indexes:

```
dbanalyzer -uUSRNAME -pUSRPASS --analysis --statistics DBNAME
```

For all tables in a particular schema SCHNAME in the previous example:

```
dbanalyzer -uUSRNAME -pUSRPASS -cSCHNAME -a -q DBNAME
```

For a particular table TBLNAME in schema SCHNAME, in database DBNAME, that user USRNAME has access to:

```
dbanalyzer -uUSRNAME -pUSRPASS -cSCHNAME -tTBLNAME -a -q DBNAME
```

One may also analyze the tables and indexes that a program ident PROGNAME with password PROGPASS has access to:

```
dbanalyzer -uUSRNAME -pUSRPASS -ePROGNAME -iPROGPASS -cSCHNAME -tTBLNAME -a
-g DBNAME
```

VMS: On VMS, arguments are converted to lower case by default. To retain upper case characters, enclose the argument in double quotes, e.g. `-p"SecretPwd"`.

An example with output

Consider the following table created by user SYSADM in database DB:

```
create table TT (c1 int not null, c2 int not null, c3 int,
    constraint TT_U unique (c1, c2));
create index TT_IDX1 on TT (c1, c2);
create index TT_IDX2 on TT (c3);
```

Note that table TT doesn't have a primary key, and TT_IDX1 is redundant because the unique constraint TT_U defines the same index implicitly.

Assume that 11 statements have accessed table TT, 2 of them have used index TT_IDX2, and 4 have used the implicit index created by TT_U.

Each statement may have been executed many times since the server started.

The following command is executed to analyze the table TT and report the usage of the table and its indexes:

```
dbanalyzer DB -u SYSADM -p SYSADM -s SYSADM --table=TT --analysis
```

The output of the example command looks as follows:

```
****Index and unique constraints optimization suggestions*****
Start analyzing table SYSADM.TT
  Used by 11 statements since server started
  Unique constraints:
    Unique constraint TT_U(c1, c2)
        Implicit index used by 4 statements since server started
  No redundant unique constraints are found
  No foreign keys are found
  Indexes:
    Index TT_IDX2(c3)
        Index used by 2 statements since server started
    Redundant index TT_IDX1(c1, c2)
        Index used by 0 statements since server started
        Duplicate with TT_U. To remove it, use the following SQL:
        DROP INDEX TT_IDX1 RESTRICT;
  Primary key is not defined. You may change an existing unique constraint or
index to primary key with the following SQL:
    ALTER TABLE TT DROP CONSTRAINT TT_U;
    ALTER TABLE TT ADD CONSTRAINT PK_TT PRIMARY KEY (c1,c2);

================================================================================
Usage statistics report created at 2022-05-12 15:44:01
Indexes are printed following their respective tables
--------------------------------------------------------------------------------
Schema Name             Object Name             Type        Use Count

SYSADM                  TT                      Table       11
SYSADM                  TT_U                    Index       4
SYSADM                  TT_IDX1                 Index       0
SYSADM                  TT_IDX2                 Index       2
================================================================================
```

In this example, `dbanalyzer` lists the implicit and explicit indexes for the table SYSADM.TT, and checks if any of them are redundant. It identifies TT_IDX1 as a duplicate index of the unique constraint TT_U, and provides SQL statements to remove this index.

`dbanalyzer` identifies that no primary key is defined for the table. It suggests to use the columns of an existing unique constraints as the primary key.

The tool reports how many statements have used each table and index. These numbers are also summarized in a table in the end of the report.

Notes

A user or program ident can only see analysis of tables and indexes which it has `SELECT` privilege to. However, the summarized usage statistics of each table/index contain the statements used by all users and programs.

Only statistics for statements retained by the server is kept.

Appendix A

Executing in Single-user Mode

Usually, users access a Mimer SQL database via a database server (multi-user mode), but in some situations it may be necessary to restrict use of a database to a single user.

Any local database can be opened in single-user mode, provided there is no database server currently running against the database.

Note: An application started in single-user mode will access the databank files directly from within its own process.
This means that the operating system user who is running the application must have access, at the operating system level, to all the existing databank files.
All new databank files created in single-user mode will typically be owned in the operating system by that user.

File Protection in Single- and Multi-user Mode

If a database which has been created in single-user mode is to be used by a database server, or vice versa, certain precautions must be observed with regard to the databank files:

- Files created in single-user mode must be accessible for read and write by a database server if the database is to be subsequently used in multi-user mode, since databank file access is performed by the database server process. The creator of the files should change the protection if necessary. Suitably the database server should have exclusive access to the databank files.
- Conversely, files created in multi-user mode are created by the database server process and will not be accessible by a specific user who needs to access the database in single-user mode. The protection on these files can only be changed by an operating system user who has privileges equivalent to those of the database server process.

Note: Individual users should not generally have direct access to databank files.

Specifying Single-user Mode Access

If the database is to be accessed in single-user mode by default, the environmental variable or logical name called `MIMER_MODE` should be defined as `SINGLE`, as shown in the examples that follow.

If MIMER_MODE is not defined or is set to MULTI, or the database is a remote one, it will be accessed in multi-user mode by default.

If MIMER_MODE is set to SINGLE and the default database, see *The Default Database* on page *37*, is set to point to a local database, the database will be opened in single-user mode. (Remote databases will be accessible through the client/server interfaces in multi-user mode).

Note: Many of the programs which are part of the Mimer SQL distribution support the command-line flags -s and -m (or /SINGLE, /MULTI) which control whether they access a database in single-user or multi-user mode.

Accessing in Single-user Mode

Mimer SQL applications that connect to a local database server when single-user access is indicated, will dynamically include a shared library when activated. This library holds all the functionality that normally is provided by the database server program.

Example

The following example session first connects to the INVENTORY database in single-user mode and then connects to the STAFF database, administered by a running database server process.

On Linux:

```
$ MIMER_DATABASE=INVENTORY
$ MIMER_MODE=SINGLE
$ export MIMER_DATABASE MIMER_MODE
$ bsql
SQL> .
SQL> .
SQL> exit;
$ unset MIMER_MODE
$ MIMER_DATABASE=STAFF
$ export MIMER_DATABASE
$ bsql
SQL> .
SQL> .
SQL> exit;
```

On OpenVMS:

```
$ DEFINE MIMER_DATABASE INVENTORY
$ DEFINE MIMER_MODE SINGLE
$ RUN MIMER$EXE:BSQL
SQL> .
SQL> .
SQL> exit;
$ DEASSIGN MIMER_MODE
$ DEFINE MIMER_DATABASE STAFF
$ RUN MIMER$EXE:BSQL
SQL> .
SQL> .
SQL> exit;
```

On Windows:

```
C:\> SET MIMER_DATABASE=INVENTORY
C:\> SET MIMER_MODE=SINGLE
C:\> BSQL
SQL> .
SQL> .
SQL> exit;
C:\> SET MIMER_MODE=
C:\> SET MIMER_DATABASE=STAFF
C:\> BSQL
SQL> .
SQL> .
SQL> exit;
```

The SINGLEDEFS Parameter File

The use of a `SINGLEDEFS` parameter file is optional.

When a single-user mode connection is established, the SQLPOOL and bufferpool data areas are dynamically created.

The SQLPOOL area will grow dynamically if more space is needed, see *SQLPOOL* on page *46*.

Linux + VMS: To change the size of the bufferpool in single-user mode a `SINGLEDEFS` file, similar to the `MULTIDEFS` file, should be created in the database home directory.

A template of this file, showing the default values for the relevant parameters, can be found in the examples directory which is located in the Mimer SQL installation directory.

```
--
--  Parameters for single-user system
--
Databanks        100   -- (40-1000)     Max number of databanks
Tables           4000  -- (500-1000000) Max number of tables
ActTrans         50    -- (10-1000000)  Max number of transactions
Pages4K          8000  -- (22-1000000)  Size of 4k bufferpool region (pages)
Pages32K         800   -- (22-1000000)  Size of 32k bufferpool region (pages)
Pages128K        96    -- (22-1000000)  Size of 128k bufferpool region (pages)
```

Note: When changing parameters in the `SINGLEDEFS` file, always change the copy in the database home directory. Never change the template file in the examples directory.

Win: The use of a `SINGLEDEFS` is not supported on the Windows platform.

Appendix B

The SQLHOSTS File on VMS and Linux

This appendix applies to the OpenVMS and Linux/macOS platforms only.

It describes the SQLHOSTS file which is used to list all the databases that are accessible to a Mimer SQL application from the node on which it resides.

For general information on how to make databases accessible, refer to *Registering the Database* on page *28*.

Linux: On a Linux node, the path name of SQLHOSTS file is /etc/sqlhosts.

The program called mimsqlhosts can be used to manage the contents of the SQLHOSTS file instead of editing it manually.

When the dbinstall program is used to install a local database on a Linux node, an entry for it is automatically added to the LOCAL section, see *LOCAL Section* on page *158*, of the SQLHOSTS file on that node.

If the file is not found, a default SQLHOSTS file is automatically generated. (See the mimhosts and sqlhosts man-pages).

VMS: On an OpenVMS node, the SQLHOSTS file can have any name and is located by translating the logical name MIMER_SQLHOSTS. The MIMSETUP command will define it to be:

```
SYS$SPECIFIC:[SYSMGR]SQLHOSTS.DAT
```

A default SQLHOSTS file is generated by the installation of the Mimer SQL software.

The SQLHOSTS File

A line of text beginning with the character sequence -- is interpreted as a comment in the SQLHOSTS file.

The SQLHOSTS file contains three sections, called: DEFAULT, LOCAL and REMOTE.

The names of the local databases on the current node are listed in the LOCAL section, see *LOCAL Section* on page *158* and the names of the remote databases accessible from the node are listed in the REMOTE section, see *REMOTE Section* on page *158*.

One of the local or remote databases can be set to be the default database for the node by specifying its name in the `DEFAULT` section, see *Default Section* on page *158*.

Database names may, in general, be up to 128 characters long and are case-insensitive.

> **VMS:** The maximum length for the name of a database on an OpenVMS node is 30 characters.

The Default SQLHOSTS File

When the Mimer SQL system is installed on a node, the following default `SQLHOSTS` file is automatically generated:

```
-- ------------------------------------------------------------------------
--
--  S Q L H O S T S
--  ===============
--
--  This file contains a list of all databases, local and remote, accessible
--  from the node where the file resides.
--
--  The DEFAULT label
--  -----------------
--  Name of default database. Can be either a REMOTE or LOCAL database name.
--  Can be overridden by setting MIMER_DATABASE to the name of a database.
--
--  The LOCAL label
--  ---------------
--  A list of all local databases on the current node, containing the
--  database name and a directory specification (Path).
--  UNIX Path -     database home, and directory path for databank lookup.
--  VMS Path -     database home.
--
--  The REMOTE label
--  ----------------
--  A list of all remote databases containing the database name, the database
--  node, the protocol to be used, the protocol interface and the protocol
--  service to be used.
--
--  Protocol, Interface and Service may be defaulted by entering ''.
--
--  Node -      network node name for computer on which the database resides.
--  Protocol -  currently tcp is supported. (tcp or '' should be specified)
--  Interface - currently not used ('' should be specified).
--  Service -   corresponds to the port number used in TCP/IP. The port number
--              Default is 1360, i.e. the port number reserved for MIMER.
--              On UNIX: The port number may either be a number or a name of a
--              service stored in the /etc/services file.
--
-- ========================================================================
DEFAULT:
--
-- Database
-- ------------------------------------------------------------------------
   example_localdb
-- ========================================================================
LOCAL:
--
-- Database           Path
-- ------------------ ------------------------------------------------------
(Linux)     SINGLE             .
(VMS)       SINGLE             SYS$DISK:[]
(Linux)     example_localdb    /directory
(VMS)       example_localdb    DISK:[DIRECTORY]
-- ========================================================================
REMOTE:
--
-- Database           Node               Protocol Interface Service
-- ------------------ ------------------ -------- --------- --------------------
   example_remotedb   server_nodename    ''       ''        1360
```

Default Section

The DEFAULT section contains a single line that specifies the default database which will be used by an application that does not explicitly specify a database to connect to, see *The Default Database* on page *37*.

The default database should be one of those listed in the LOCAL or REMOTE sections.

LOCAL Section

The LOCAL section contains a list of all the local databases residing on the current machine, see *The Local Database* on page *28*.

Each line under the LOCAL keyword should contain two fields, separated by one or more blanks or tab characters. The first field specifies the database name.

Linux: On a Linux node, the second field may be a colon (:) separated search path specification.

The first directory in the search path is taken as the database home directory and the other directories in the search path will be used to locate databank files which have a file specification stored in the data dictionary without an explicit directory.

VMS: On an OpenVMS node, the second field specifies a directory which will be the home directory for the database.

The Mimer SQL system databank SYSDB will be located in the database home directory and other databanks will typically be located relative to it, see *Locating Databank Files* on page *9*.

REMOTE Section

The REMOTE section contains a list of all accessible databases that reside on other nodes in the network environment, see *Accessing a Database Remotely* on page *29*.

Access to these databases is provided by using either DECNET or TCP/IP to establish a client/server connection to the remote machine.

Each entry in the REMOTE section contains up to five fields, separated by spaces and/or tab characters.

The DATABASE field specifies the name of the remote database.

The NODE field should specify the network node name of the remote machine. If the TCP/IP interface is used, the IP address may be specified here.

Linux: The PROTOCOL field should specify tcp or two single quotation marks ''.

VMS: The PROTOCOL field may specify DECNET or TCP depending on the type of network protocol that should be used to create the client/server connection. The default, specified by two single quotation marks '', is TCP.

The INTERFACE field is currently not used. Specify '' (two single quotation marks) here.

If using TCP/IP, the SERVICE field specifies the TCP/IP port number the database server uses. The default is 1360, which has been reserved by Mimer Information Technology AB for Mimer SQL client/server communication.

Linux: When TCP/IP is used under Linux, the value in the SERVICE field may be the actual port number, the name of a service stored in the /etc/services file or two single quotation marks '' for the default value 1360.

VMS: For a Mimer SQL database server using DECNET, the SERVICE field should contain the database name, which is also the default.

The server listens to the network object using the same name as the database.

Remote Section Parameters

The remote section parameters are summarized below, depending on the protocol selected. The character sequence '' is two single quotation marks and specifies the default value for a parameter:

TCP - Linux

Parameter	Explanation
DATABASE	Remote database name
NODE	TCP/IP node name or IP number
PROTOCOL	'' or TCP
INTERFACE	''
SERVICE	TCP/IP_port_number or TCP/IP service name or ''. When '' is used to specify the default SERVICE, the TCP/IP port number 1360 will be used.

TCP - OpenVMS

Parameter	Explanation
DATABASE	Remote database name
NODE	TCP/IP node name or IP number. If the name is preceded by an '@' character, a logical name lookup will be made on the name and the translation will be used to specify the node name.
PROTOCOL	Specify either TCP or two single quotation marks ''.

Parameter	Explanation
INTERFACE	Specify either a list of connection options, separated by commas, or two single quotation marks ''. The connection options are: IP=4: Use IPv4 only IP=6: Use IPv6 only MEMBER=node: Terminate connection whenever the specified node leaves the cluster. The node name can be preceded by an '@' character which will cause a logical name lookup to be performed on the name.
SERVICE	TCP/IP port number or TCP/IP service name or ''. When '' is used to specify the default SERVICE, the TCP/IP port number 1360 will be used.

DECNET – OpenVMS Only

Parameter	Explanation
DATABASE	Remote database name
NODE	Decnet node name
PROTOCOL	DECNET
INTERFACE	''
SERVICE	Decnet network object or ''. When '' is used to specify the default SERVICE, the value of remote database name will be used.

Appendix C

The MULTIDEFS File on VMS and Linux

This appendix applies to the OpenVMS and Linux (including macOS) platforms only.

It describes the `MULTIDEFS` file which forms part of a local database definition, see *The Local Database* on page *28*, for a database residing on an OpenVMS or Linux node.

This file contains operational parameters for the database server for such a database and these are read when the database server is started. It is not possible to change the parameters for a running database server.

The MULTIDEFS Parameter File

Comments in `MULTIDEFS` are introduced by the character sequence `--`, or by the character `!` or `#`.

A new MULTIDEFS file can be generated by using the command:

```
mimcontrol -g
```

If the `MULTIDEFS` file is not found when starting a database server, the `MIMCONTROL` command will create a new file and fill it with the default values for all parameters.

Linux: On a Linux node, the `MULTIDEFS` file is located in the database home directory and is called `multidefs`.

VMS: On an OpenVMS node, the `MULTIDEFS` file is located in the database home directory and is called `MULTIDEFS.DAT`.

The actual default values used may vary and may depend on factors like machine type and the amount of physical memory available on the machine.

The following is an example of the `MULTIDEFS` parameter file which may be generated by `MIMCONTROL`:

Example of MULTIDEFS File on Linux

```
-- Mimer SQL version 11.0.5A parameters generated 2021-01-18 10:31
Databanks          100                # Max # of databanks (20-1000)
Tables             4000               # Max # of tables (500-1000000)
ActTrans           20000              # Max # of active trans (500-1000000)
SQLPool            1000               # Initial SQLPool (400-8388607 kb)
RequestThreads     8                  # # of request threads (1-100)
BackgroundThreads  3                  # # of background threads (1-100)
TcFlushThreads     1                  # # of t-cache flush threads (0-20)
Users              100                # Max # of logged in users (1-5000)
DBCheck            1                  # DB check, 0=index, 1=all, 2=immediate,
                                        3=im. index, 4=im. all (0-4)
Pages4K            206867             # # of 4K bufferpool pages (11-2147480000)
Pages32K           18784              # # of 32K bufferpool pages (7-2147480000)
Pages128K          2187               # # of 128K bufferpool pages (0-2147480000)
DelayedCommit      0                  # Delayed commit, 0=Off 1=On
                                        2=Disabled (0-2)
DelayedCommitTimeout 100              # Delayed commit timeout in milliseconds
                                        (0-60000)
GroupCommitTimeout 2                  # Group commit timeout in milliseconds
                                        (0-20)
Oper                                  # Receivers for messages
DumpPath           .                  # Path for dump directory
TCPPort            inetd              # TCP/IP port
MaxSQLPool         216000             # SQLPool max size (2400-16777215 kb)
NetworkEncryption  1                  # Client/server encryption, 0=None
                                        1=Optional, 2=Required (0-2)
MemLock            0                  # Lock bpool in memory, 0=No 1=Yes (0-1)
MiniDump           1                  # Small bufferpool dump (no page content),
                                        0=No 1=Yes (0-1)
BackgroundPriority 0                  # Thread priority, 0=Default, 1=Highest,
                                        40=Lowest (0-40)
AutoStart          1                  # Autostart, 0=No, 1=Yes (0-1)
DumpScript         ./.dumper.sh %p    # Dump Script
ServerType         3                  # Server type: 3=mimexper, 7=miminm (3-9)
IOQueue          128                  # Max # of concurrent I/O requests
                                        (0-65535)
```

Example of MULTIDEFS File on OpenVMS

```
-- Mimer SQL version 11.0.3C Beta Test parameters generated 2020-05-03 22:30
Databanks           100                 # Max # of databanks (20-1000)
Tables              4000                # Max # of tables (500-1000000)
ActTrans            20000               # Max # of active trans (500-1000000)
SQLPool             1000                # Initial SQLPool (400-8388607 kb)
RequestThreads      8                   # # of request threads (1-100)
BackgroundThreads   3                   # # of background threads (1-100)
TcFlushThreads      1                   # # of t-cache flush threads (0-20)
Users               100                 # Max # of logged in users (1-5000)
DBCheck             1                   # DB check, 0=index, 1=all, 2=immediate,
                                          3=im. index, 4=im. all (0-4)
Pages4K             206867              # # of 4K bufferpool pages (11-33554432)
Pages32K            18784               # # of 32K bufferpool pages (7-4194304)
Pages128K           2187                # # of 128K bufferpool pages (0-1048576)
DelayedCommit       2                   # Delayed commit, 0=Off 1=On 2=Disabled
                                          (0-2)
DelayedCommitTimeout 100                # Delayed commit timeout in
                                          milliseconds (0-60000)
GroupCommitTimeout  2                   # Group commit timeout in milliseconds
                                          (0-20)
Oper                OPER                # Receivers for messages
DumpPath            <>                  # Path for dump directory
TCPPort             1360                # TCP/IP port
MaxSQLPool          216000              # SQLPool max size (2400-16777215 kb)
MemLock             0                   # Lock bpool in memory, 0=No 1=Yes (0-1)
MiniDump            1                   # Small bufferpool dump (no page content)
                                          , 0=No 1=Yes (0-1)
DECPort             MULTI               # Decnet network object
ProcName            MULTI               # Process name prefix
NetUsers            5000                # Max # of network users (1-5000)
ServPrio            5                   # VMS prio for server process (0-16)
Cleanup             60                  # Cleanup interval (1-10000 seconds)
Multithread         0                   # Kernel thread limit (0-64)
BPResident                              # Bufferpool resident area name
HomeRAD             -                   # Home RAD
```

MULTIDEFS Parameters

Parameter	Definition
`Databanks`	Specifies the maximum number of databank files that the database server can have open at any one time.
`Tables`	Specifies the maximum number of tables that can be accessed simultaneously by the database server.
`ActTrans`	Specifies the maximum number of transactions that can be active in the database server.
`SQLPool`	Initial size of the SQLPool area in K bytes. This area contains information about each session, i.e. opened tables and databanks, compiled SQL programs, etc. The SQLPool area will expand automatically if it is too small, but it will not be larger than `MaxSQLPool`.
`RequestThreads`	The number of threads in the database server that can serve client requests.
`BackgroundThreads`	The number of background threads in the database server.

Parameter	Definition
`TcFlushThreads`	Extra threads that run in the background to help clear the transaction cache. This is beneficial for systems with long-running transactions. The thread keeps the size down of the transaction cache by deleting records that are no longer used. When there are no long running transactions the cache can be cleared efficiently without scanning the cache so in this case the thread is not needed. Default is 1 thread. To get the same behavior as in version 10.0 specify 0 threads for this parameter. For **very** large databases with long-running transactions more than 1 thread can be used.
`Users`	The maximum number of users that are allowed to connect to the database server. This parameter should not exceed the number of users specified in the Mimer SQL license key. This number is also used to calculate the size of the shared memory region used for local database server communication. About 70 Kbytes of shared memory will be allocated for each user.

Parameter	Definition
DBCheck	A number which specifies what kind of check that should be performed when a databank is opened which previously was not closed properly. **0 - check index pages** Index pages only are checked in the foreground while applications that access the databank waits for the operation to complete. **1 - check data pages** A full databank check (involving index and data pages) provides for more secure operations, but may take much longer to execute than an index page check. When a full check is done, the index pages are checked in the foreground and the data pages are checked in the background so there is a smaller effect on performance. **2 - Immediate restart, no check** This options performs no checking when the file is opened. The system still verifies the integrity of each page through a checksum. A few pages may have been pre-allocated and these are not reclaimed when this option is used. If the option is subsequently changed these pages will be reclaimed the next time the databank is opened. **3 - Immediate restart, check index pages** This option performs a check of all index pages in the databank in the background. This is done concurrently with other operations on the system. **4 - Immediate restart, check all pages** This option performs a check of all pages in the databank in the background. This is done concurrently with other operations on the system. The Immediate restart options require a license key called “Imm Restart”. Databank checks can be avoided by always shutting down the database server properly with the MIMCONTROL command, especially prior to shutting down the machine.

Parameter	Definition
`Pages4K`	The number of 4 Kbytes pages in the bufferpool area containing pages from the databank files. The default value of this parameter is 12.5% of the total RAM memory in the machine. **VMS:** There may be an OpenVMS limit set for the amount of memory a process may allocate, this limit will not be exceeded. Among the various OpenVMS parameters, WSMAX is likely to be of primary interest in connection with this limit.
`Pages32K`	The number of 32 Kbytes pages in the bufferpool area containing pages from the databank files. The default value of this parameter is 8.33% of the total RAM memory in the machine.
`Pages128K`	The number of 128 Kbytes pages in the bufferpool area containing pages from the databank files. The default value of this parameter is 5% of the total RAM memory in the machine.

Parameter	Definition
`DelayedCommit`	This option controls how quickly a transaction commit is secured on disk. It greatly affects the performance of the database server. For example, if a single user commits two transactions in quick sequence the database server may use a single I/O to secure both transactions when delayed commit is on. Transactions are never reordered by using the delayed commit option. I.e. it is not possible for a later transaction to be secured on disk before an earlier one. The database is thus always returned to a consistent state after a machine crash. However, if a transaction has been committed but not yet written to disk it will be lost if the database server or machine goes down in an uncontrolled fashion. Transactions that use the XA transaction protocol are automatically committed with delay commit disabled. Delayed commit option can be set to one of the following: **0 - Default off** In this mode delayed commit is not used unless a transaction is set to use delayed commit by the application. This is the default. **1 - Default on** In this mode all transactions where the delay mode has not been explicitly set are delayed. The transaction will be secured within the time-out period specified. If other transactions are committed before the time-out occurs the transactions may be combined into a single I/O to boost performance. **2 - Disabled** In this mode all transactions are secured to disk immediately and the application will not regain control after a commit until the transaction has been secured. This option overrides any application settings for delay commit.
`DelayedCommitTimeout`	This specifies the number of milliseconds to wait before the transaction is written to disk. If a value of zero is specified transactions are not flushed until the server determines that the commit set page is full. In general, this is not recommended as transactions are likely to be lost if there is an uncontrolled machine stop. Default is 100 milliseconds.

Parameter	Definition
`EnablePasswords`	**VMS:** Enable password login: 0 - password login disabled. (Only `OS_USER` login possible.) 1 - password login allowed.
`GroupCommitTimeout`	How many milliseconds to wait for other transactions to commit before proceeding with first transaction. If another transaction arrives within the timeout period if will be grouped with existing transactions before they are committed together with a single I/O rather. This improves overall performance but the delay prolongs commits time on a system with low load. Default is one millisecond.
`Oper`	This parameter gives a list of host system users, i.e. operators, or e-mail addresses that should receive e-mail notification of serious problems with the database server. **VMS:** On OpenVMS, you can also specify `OPER`. This will enable that notification messages are sent to the central operator, i.e. processes that have set: `$ REPLY/ENABLE=CENTRAL.`
`DumpPath`	This parameter may specify an alternate path for the dump directories. The default is to create dump directories under the database home directory.
`TCPPort`	Specifies how the database server should handle incoming TCP/IP connection requests. If this parameter is set to - (a single dash), the TCP/IP capability will be disabled for the database server. **Linux:** On Linux, the `TCPPort` parameter is, by default, set to `inetd` which means that the TCP/IP port server program, `mimtcp`, will be used for establishing a connection to any Mimer SQL database server (of version 8 and later). In this case clients may connect to the port to which `mimtcp` listens, usually 1360, and the handshake will be passed over to the requested Mimer SQL database server. If a TCP/IP port number is specified, the database server will listen directly to that port. **VMS:** On OpenVMS, the `TCPPort` parameter is, by default, set to the TCP/IP port number `1360`. The TCP/IP port server program, `MIMTCP`, will automatically be started to listen to the given port, serving all Mimer SQL database servers (of version 8 and later) set up to use that port.

Parameter	Definition
MaxSQLPool	The maximum size (in kilobytes) of the SQLPool. The SQLPool memory area grows dynamically, but the size will never exceed this parameter. Use this parameter to control the maximum virtual size (maximum page file usage) for the database server process.
NetworkEncryption	Controls the use of encryption of network communication over TCP/IP between server and clients. **0 = Network encryption disabled** Network encryption is not supported or not used. **1 = Network encryption preferred** Network encryption is enabled for version 11 clients. Older clients use unencrypted network communication. When this setting is used, older clients without support for network encryption are allowed to communicate with the database server over TCP/IP. Use this option when there is a mix of older and newer clients that communicate with the database server over TCP/IP. This is the default value. **2 = Network encryption required** The database server requires all clients to use encrypted communication when communicating over TCP/IP. Clients that do not support encryption are rejected at login with error code -18531. Named Pipes via OS-user login is not allowed. This option is recommended over option 1 when possible (i.e. when there are no older clients that need to be supported.)
MemLock	A number which specifies whether the bufferpool and communication buffers should be locked in memory (1) or not locked in memory (0).
Minidump	Small bufferpool dump (no page content). 0 = No 1 = Yes (default)

Parameter	Definition
BackgroundPriority	**Linux:** Specifies if the background threads should run at a higher priority than other server threads. Default is 0 meaning that the priority is not changed. Valid values are between 1 and 40, where 1 is the highest priority and 40 the lowest. A priority of 20 will give the same priority as the default value. During certain circumstances like in situations where the background threads cannot manage to shorten a transaction queue a higher priority might help. Giving a too high priority might have unexpected side effects. To be able to change the thread priority Linux capabilities is used. To allow the Mimer SQL executable to do this the `setcap` command is used: `sudo setcap CAP_SYS_NICE+iep /opt/mimersql1103-11.0.3D/bin/mimexper` To do this the `libpam-cap` needs to be installed, and the user that will manage the Mimer SQL database must be give permissions to change priorities. This is done by adding `CAP_SYS_NICE <user>` after the line that says `none *` in `/etc/security/capability.conf`.
AutoStart	**Linux:** By default, this parameter is set to 1 which indicates that the database should be started automatically when the operating system goes into multi-user mode. If the parameter is set to 0 the database will not be started automatically.
DumpScript	**Linux:** If the database server goes into an erroneous and unrecoverable state, it will produce dumps of the current internal database structures before it goes down. If this situation occurs, it is of great importance for the error detection process to get a Linux kernel stack trace from the location where the error was located. By defining this parameter to a command that can produce a kernel trace, such as `pstack`, stack information will be automatically generated to `mimer.log`. The `%p` option, used in the example above, is used to get the current process ID as a parameter to the command given.

Parameter	Definition
DECPort	**VMS:** Specifies the DECNET network object that the database server listens to. Each database server must use a unique network object. The default value is the database name. If you set this parameter to - (a single dash), the DECNET capability will be disabled for the database server.
ProcName	**VMS:** This parameter specifies the process name prefix for the database server. (The last part of the process name is always Srv). Specify a maximum of 11 characters. The default value is to use the first 11 characters of the database name.
NetUsers	**VMS:** This parameter specifies the number of users who can access the database through a network connection. The value used by the system will be the minimum of this parameter and the Users parameter. The default value is 5000. Since the Users parameter can not be larger than 5000, this means that all users may be network users.
ServPrio	**VMS:** This parameter specifies the OpenVMS priority for the database server process.
Cleanup	**VMS:** Specifies the time (in seconds) between the cleanup sweeps that check for terminated database clients.
BPResident	**VMS:** If the BPResident parameter is blank (default) the bufferpool will be allocated in normal process memory and is backed by the paging file. The paging file process quota for the database server will be increased accordingly. If the BPResident parameter specifies a name, a memory resident global section with this name will be created to hold the buffer pool. Since physical memory is used, the buffer pool will not be backed by the paging file. Also, the working set quota of the process does not have to include the buffer pool. This is recommended for larger buffer pools. To use the BPResident parameter, the user that starts the database server must hold the VMS$MEM_RESIDENT_USER process right. Please see the VMS Guide for more information.

Parameter	Definition
`Multithread`	**VMS:** Multithread can be used to limit the number of kernel threads used by a database server. When running several instances of Mimer servers on a machine with a large number of cores, it can be beneficial to limit the number of kernel threads (and the number of cores) each database server can use. The default value is zero, which means that the number of kernel threads are not limited. This feature requires OpenVMS 8.4 or later.
`HomeRAD`	**VMS:** The parameter HomeRAD can be used to specify a Home RAD (Resource Affinity Domain) for the database server process. If a server hosts multiple Mimer database servers, it can be beneficial to start the servers in different RAD's. An article in OpenVMS technical Journal V16 about RAD support on Integrity servers can be found here: https://h41379.www4.hpe.com/openvms/journal/v16/rad.pdf The default value of the HomeRAD parameter is - (a minus sign), which means that no HomeRAD is set. By specifying a number, the database server process will use the specified RAD as its home RAD. If you specify a Home RAD and also specify a memory resident global section (parameter BPResident), you should create the section in the same RAD.
`ServerType`	This option decides which Mimer SQL database server program that should be started to operate the database files for the database: **3 - mimexper** The Mimer SQL Experience database server. This is the standard database server. See *Mimer SQL Experience Database Server* on page *43*. **7- miminm** The Mimer SQL In-memory database server. See *Mimer SQL In-memory Database Server* on page *44*.

Parameter	Definition
`IOQueue`	**Linux:** Specifies the maximum number of concurrent IO requests queued to the operating system. Default is 128, but more advanced disk systems such as SANs, battery backed caching IO controllers, PCI Express connected SSDs and NVMe SSDs can make use of larger queues. This can give a significantly higher database performance. However, specifying a too large queue can overload the IO subsystems. Maximum queue length is 65535.

Appendix D

Data Dictionary Tables

This appendix documents the organization of the data dictionary tables, which are stored in the databank SYSDB.

The tables are created by the SDBGEN program when the system is created. The tables are created in a schema called SYSTEM and are thus effectively owned by a pseudo ident called SYSTEM.

The database administration ident SYSADM is granted SELECT access on the dictionary tables with the WITH GRANT OPTION. No other user may read the data dictionary base tables unless authorized to do so by SYSADM.

A set of system views is defined on the data dictionary tables when the system is installed, see the *Mimer SQL Reference Manual, Chapter 13, Data Dictionary Views* for more information).

The logical group PUBLIC is granted SELECT access on these views, so that any user may read the dictionary information presented in the views. Many of the view definitions restrict the information presented to descriptions of objects and privileges accessible to the current user.

Note: If SYSADM reads the contents of such a view, the result shows only the objects and privileges to which SYSADM has access. In order to gain information on inaccessible objects and privileges, SYSADM must read the contents of the dictionary base tables directly.

The SYSADM ident may define installation-specific views on the data dictionary tables to supplement the system-defined views. Such views may be tailor-made for the installation or system in use, and SELECT access on the views may be granted to limited user groups if desired.

All maintenance of the data dictionary is performed by internal routines and is invisible to the user. No user, including SYSADM, may alter the contents of the data dictionary directly.

Note: Mimer reserves the right to change the internal organization of the data dictionary, without maintaining backward compatibility with user-written application programs which read the data dictionary tables directly.

Summary of Data Dictionary Tables

Table name	Description
SYSTEM.API_FUNCTION	Translation of id to function or module name.
SYSTEM.AST_CODES	Binary representation of the search condition of views in the database (for internal use).
SYSTEM.AST_SOURCES	Textual representation of view definitions and tables and domains with check constraints.
SYSTEM.ATTRIBUTES	Attributes of user-defined types.
SYSTEM.CHAR_SETS	Character sets in the database.
SYSTEM.CHECK_CONSTRAINTS	Check constraints defined for tables and domains.
SYSTEM.COLLATE_DEFS	Collation definitions.
SYSTEM.COLLATIONS	Character collations in the database.
SYSTEM.COLUMN_OBJECT_USE	Columns that depend on other database objects.
SYSTEM.COLUMN_PRIVILEGES	Instances of privileges granted on a column.
SYSTEM.COLUMNS	Columns in tables or views.
SYSTEM.COMMENTS	Comments on objects.
SYSTEM.DATABANKS	Databanks in the database.
SYSTEM.DIRECT_SUPERTYPES	Direct supertypes in the database.
SYSTEM.DOMAIN_CONSTRAINTS	Constraints defined for domains.
SYSTEM.DOMAINS	Domains in the database.
SYSTEM.EXEC_STATEMENTS	Precompiled statements.
SYSTEM.FIPS_FEATURES	Details about all FIPS features.
SYSTEM.FIPS_SIZING	Details about FIPS limits.
SYSTEM.HEURISTICS	Details about heuristics.
SYSTEM.KEY_COLUMN_USAGE	Columns in an index or in a primary, unique or foreign key.

Table name	Description
SYSTEM.LEVEL2_RESTRICT	Restrictions for domains legal for use by DB level 2.
SYSTEM.LEVEL2_VIEWCOL	Columns of views acceptable by DB level 2.
SYSTEM.LEVEL2_VIEWRES	Restrictions for DB level 2.
SYSTEM.LIBRARIES	External libraries.
SYSTEM.LOGINS	OS_USER logins for user idents.
SYSTEM.MANYROWS	Dummy table with more than one row.
SYSTEM.MESSAGE	Translation of message code to message text.
SYSTEM.METHOD_SPECIFICATION_PARAMETERS	Method specification parameters.
SYSTEM.METHOD_SPECIFICATIONS	Method specifications.
SYSTEM.MODULES	SQL-server modules in the database.
SYSTEM.NANO_DATABANKS	(Currently not used.)
SYSTEM.NANO_DESCRIPTORS	(Currently not used.)
SYSTEM.NANO_OBJECTS	(Currently not used.)
SYSTEM.NANO_ROUTINE_USE	(Currently not used.)
SYSTEM.NANO_USERS	(Currently not used.)
SYSTEM.OBJECT_COLUMN_USE	Columns referenced by other database objects.
SYSTEM.OBJECT_OBJECT_USE	Objects that have a dependency on another object.
SYSTEM.OBJECT_PROGRAMS	Information about predefined executable statements for table operations and routines.
SYSTEM.OBJECTS	Objects in the database.
SYSTEM.ONEROW	Dummy table containing one row.
SYSTEM.PARAMETERS	Parameters of routines in the database.
SYSTEM.REFER_CONSTRAINTS	Referential constraints in the database.

Table name	Description
SYSTEM.ROUTINES	Procedures and user-defined functions in the database.
SYSTEM.SCHEMATA	Schemas in the database.
SYSTEM.SEQUENCE_VALUE_TABLE	Current values of sequences.
SYSTEM.SEQUENCES	Sequences in the database.
SYSTEM.SERVER_INFO	Attributes of the current database system or server.
SYSTEM.SEVERITY	Severity levels and optional module for error codes.
SYSTEM.SOURCE_DEFINITION	Source definitions for defaults, check constraints, views, modules, procedures, functions and triggers.
SYSTEM.SPECIFIC_NAMES	Specific names of the routines in the database.
SYSTEM.SQL_CONFORMANCE	SQL functionality.
SYSTEM.SQL_LANGUAGES	SQL standards and SQL dialects supported.
SYSTEM.STATEMENT_DESCRIPTORS	Compiled code for statements.
SYSTEM.STATEMENT_ROUTINE_USE	Compiled routine used by precompiled statements.
SYSTEM.SYNONYMS	Synonyms and shadows in the database.
SYSTEM.TABLE_CONSTRAINTS	Table constraints in the database.
SYSTEM.TABLE_PRIVILEGES	Instances of privileges granted on a table.
SYSTEM.TABLE_TYPES	The types of table supported.
SYSTEM.TABLES	Tables and views in the database.
SYSTEM.TRANSLATIONS	Character translations in the database.
SYSTEM.TRIGGERED_COLUMNS	Table columns explicitly specified in an UPDATE trigger event.
SYSTEM.TRIGGERS	Triggers in the database.
SYSTEM.TYPE_INFO	Information about data types supported.

Table name	Description
SYSTEM.USAGE_PRIVILEGES	Instances of privileges which grant the right to use a database object.
SYSTEM.USER_DEF_TYPES	User-defined types in the database.
SYSTEM.USERS	Idents (GROUP, PROGRAM or USER) in the database.
SYSTEM.VIEWS	Views in the database.

SYSTEM.API_FUNCTION

Records translations of id to function or module name.

Column name	Data type	Description
MODULEID	INTEGER	System identifier for module.
API_FUNCTION	INTEGER	System identifier for function.
TEXT	CHAR(40)	Name of the function or module.

Primary key: MODULEID, API_FUNCTION

SYSTEM.AST_CODES

Records the binary representation of the search condition of views in the database (for internal use).

Column name	Data type	Description
AST_SYSID	INTEGER	System identifier for the view.
AST_VERSION	INTEGER	Current AST-revision version number.
STRUCT_VERSION	INTEGER	Compiler version number.
SEQUENCE_NO	INTEGER	Sequence number within representation.
AST_LENGTH	INTEGER	Length of binary data in AST_CODE.
AST_CODE	VARCHAR(1200)	Binary representation of the view search condition.

Primary key: AST_SYSID, AST_VERSION, STRUCT_VERSION, SEQUENCE_NO

SYSTEM.AST_SOURCES

Records the textual representation of view definitions and domains or tables with check constraints in the database (for internal use).

Column name	Data type	Description
ASTS_SYSID	INTEGER	System identifier for the object which the definition represents.
SEQUENCE_NO	INTEGER	Sequence number within representation.
ASTS_LENGTH	INTEGER	Length of representation.
ASTS_SOURCE	NCHAR VARYING(400)	Textual representation of view, domain or table.

Primary key: ASTS_SYSID, SEQUENCE_NO

SYSTEM.ATTRIBUTES

Records user-defined type attributes.

Column name	Data type	Description
UDT_SYSID	INTEGER	System identifier for the user-defined type, to which the attribute belongs.
ATTRIBUTE_ID	INTEGER	System identifier for the attribute.
ATTRIBUTE_NAME	NCHAR_VARYING(128)	Attribute name.
ORDINAL_POSITION	INTEGER	The ordinal position of the attribute in the user-defined type. The first attribute is number 1.
COLLATION_SYSID	INTEGER	System identifier for the collation used by the attribute.
CHARSET_SYSID	INTEGER	System identifier for the attribute's character set.
ATTRIBUTE_UDT_SYSID	INTEGER	System identifier for the attribute's data type, if user-defined type.

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	The data type of the attribute. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier for data type.

Column name	Data type	Description
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	This defines the total number of significant digits to the right of the decimal point. For INTEGER and SMALLINT, this is 0. For CHARACTER, VARCHAR, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, it is the null value.

Column name	Data type	Description
NUMERIC_PREC_RADIX	INTEGER	For numeric data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_OFFSET	INTEGER	Internal offset for attribute in type.
IS_NULLABLE	BOOLEAN	Nullability attribute TRUE = The attribute can be null FALSE = The attribute can not be null
OBSERVER_METHOD	INTEGER	System identifier for observer method for attribute.
MUTATOR_METHOD	INTEGER	System identifier for mutator method for attribute.

Column name	Data type	Description
OBSERVER_METHOD_ SPECIFICATION	INTEGER	System identifier for observer method specification for attribute.
MUTATOR_METHOD_ SPECIFICATION	INTEGER	System identifier for mutator method specification for attribute.
INTERNAL_LENGTH	INTEGER	Internal length in bytes of attribute.
AS_LOCATOR	BOOLEAN	One of: TRUE = declared as locator FALSE = not locator

Primary key: UDT_SYSID, ATTRIBUTE_ID

Unique constraint: UDT_SYSID, ORDINAL_POSITION

Unique constraint: UDT_SYSID, ATTRIBUTE_NAME

SYSTEM.CHAR_SETS

Records character sets in the database.

Column name	Data type	Description
CHARSET_SYSID	INTEGER	The system identifier for the character set.
FORM_OF_USE	VARCHAR(128)	A user-defined name that indicates the form-of-use of the character set.
NUMBER_OF_CHARS	INTEGER	Number of characters in the character set.
DEF_COLLATE_SYSID	INTEGER	System identifier for the character collation.

Primary key: CHARSET_SYSID

SYSTEM.CHECK_CONSTRAINTS

Records check constraints defined for tables and domains in the database.

Column name	Data type	Description
CONSTRAINT_SYSID	INTEGER	System identifier for the check constraint.
CHECK_CLAUSE	NCHAR VARYING(200)	The text of the search condition of the check constraint. If the text is too long or null value, it will be stored in the SOURCE_DEFINITION table.

Primary key: CONSTRAINT_SYSID

SYSTEM.COLLATE_DEFS

Records collation definitions in the database.

Column name	Data type	Description
COLLATION_SYSID	INTEGER	System identifier for the collation.
COLLATION_SEQNO	INTEGER	Sequence number for each row of a definition (starts by 1).
COLLATION_TYPE	INTEGER	Internal identification for the base of the collation.
FROM_COLLATION_SYSID	INTEGER	System identifier for the collation on which this collation is based.
COLLATION_DEF_CHAR_LENGTH	INTEGER	Total length of the collation definition in number of characters.
COLLATION_DEF_CHAR_OFFSET	INTEGER	Offset to where the delta string for this collation starts in the definition.
COLLATION_DEF	NCHAR VARYING(400)	Complete collation definition for this collation (i.e. including the base collation).

Primary key: COLLATION_SYSID, COLLATION_SEQNO

SYSTEM.COLLATIONS

Records character collations in the database.

Column name	Data type	Description
COLLATION_SYSID	INTEGER	System identifier for the collation.
CHARSET_SYSID	INTEGER	System identifier for the character set.
PAD_ATTRIBUTE	VARCHAR(20)	One of the following values: NO PAD = the collation has the no pad attribute PAD SPACE = the collation has the pad space attribute.

Primary key: COLLATION_SYSID

SYSTEM.COLUMNS

Records table and view columns in the database.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier for the table or view.
COLUMN_ID	INTEGER	Column identifier.
COLUMN_NAME	NCHAR VARYING(128)	The name of the table or view column.
ORDINAL_POSITION	INTEGER	The ordinal position of the column in the table. The first column in the table is number 1.
DOMAIN_SYSID	INTEGER	System identifier for the domain used by the column.
COLLATION_SYSID	INTEGER	System identifier for the collation used by the column.

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	Identifies the data type of the column. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier for the internal data type.
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.

Column name	Data type	Description
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	For NUMERIC data types, this shows the total number of significant digits to the right of the decimal point. For INTEGER values this is 0. For all other types, it is the null value.
NUMERIC_PREC_RADIX	INTEGER	For NUMERIC data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and interval data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.

Column name	Data type	Description
IS_NULLABLE	BOOLEAN	One of: FALSE = the column is not nullable, according to the rules in the international standard TRUE = the null value is allowed in the column.
IS_UPDATABLE	BOOLEAN	One of: FALSE = the column is not updatable TRUE = the column is updatable.
IS_INSTEAD_OF	BOOLEAN	One of: FALSE = the updatability of the column does not depend on an instead of trigger TRUE = the updatability of the column depends on an instead of trigger.
INTERNAL_LENGTH	INTEGER	Internal length in bytes of value.
INTERNAL_INDEX	INTEGER	Internal index for value in record.
INTERNAL_OFFSET	INTEGER	Internal offset for value in record.
INTERNAL_VC_OFFSET	INTEGER	Internal offset for field containing actual length for a varying length item.
CHARSET_SYSID	INTEGER	System identifier for the character set used by the column.
UDT_SYSID	INTEGER	System identifier for a user-defined type.
COLUMN_CARD	BIGINT	Number of unique values in column.
COLUMN_CARD_NONULL	BIGINT	Number of values in column that are not null.
COLUMN_LOW_VALUE	BINARY(16)	Lowest value in column.
COLUMN_HIGH_VALUE	BINARY(16)	Highest value in column.

Column name	Data type	Description
COLUMN_DEFAULT	NCHAR VARYING(200)	This shows the default value for the column. If the default value is a CHARACTER string, the value shown is the string enclosed in single quotes. If the default value is a NUMERIC LITERAL, the value is shown in its original character representation without enclosing quotes. If the default value is a DATE, TIME or TIMESTAMP, the value shown is the appropriate keyword (e.g. DATE) followed by the literal representation of the value enclosed in single quotes, see the *Mimer SQL Reference Manual* for a description of DATE, TIME and TIMESTAMP literals). If the default value is a pseudo-literal, the value shown is the appropriate keyword (e.g. CURRENT_DATE) without enclosing quotes. If the default value is the null value, the value shown is the keyword NULL without enclosing quotes. If the default value cannot be represented without truncation, then a zero-length string is stored. If no default value was specified then its value is the null value. The value of COLUMN_DEF is syntactically suitable for use in specifying default-value in a CREATE TABLE or ALTER TABLE statement.

Primary key: TABLE_SYSID, COLUMN_ID

Unique constraint: COLUMN_NAME, TABLE_SYSID

Unique constraint: TABLE_SYSID, ORDINAL_POSITION

Secondary index: TABLE_SYSID, INTERNAL_INDEX, INTERNAL_OFFSET

SYSTEM.COLUMN_OBJECT_USE

Records table or view columns that depend on other objects in the database.

Column name	Data type	Description
OBJECT_SYSID	INTEGER	System identifier for the object on which the column depends.
TABLE_SYSID	INTEGER	System identifier for the table or view.
COLUMN_ID	INTEGER	Id for the column.

Primary key: OBJECT_SYSID, TABLE_SYSID, COLUMN_ID

Secondary index: TABLE_SYSID, COLUMN_ID

SYSTEM.COLUMN_PRIVILEGES

Records instances of privileges granted on a column.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier for the table or view.
COLUMN_ID	INTEGER	The id of the table or view column.
PRIVILEGE_TYPE	VARCHAR(20)	One of: INSERT REFERENCES SELECT UPDATE.
GRANTEE_SYSID	INTEGER	The sysid of the ident to whom the privilege was granted.
GRANTOR_SYSID	INTEGER	The sysid of the ident granting the privilege.
IS_GRANTABLE	BOOLEAN	One of: FALSE = WITH GRANT OPTION is not held for the privilege TRUE = WITH GRANT OPTION is held for the privilege.
IS_INSTEAD_OF	BOOLEAN	One of: FALSE = The privilege has been granted explicitly TRUE = The privilege is granted because an instead of trigger has been defined on the view to which the column belongs.

Primary key: TABLE_SYSID, COLUMN_ID, PRIVILEGE_TYPE, GRANTEE_SYSID, GRANTOR_SYSID

Secondary index: GRANTEE_SYSID

Secondary index: GRANTOR_SYSID

SYSTEM.COMMENTS

Comments on objects in the database.

Column name	Data type	Description
OBJECT_SYSID	INTEGER	System identifier for object.
COLUMN_ID	INTEGER	Column ordinal position, 0 if comment is not on column.
COMMENT	NCHAR VARYING(254)	Comment.

Primary key: OBJECT_SYSID, COLUMN_ID

SYSTEM.DATABANKS

Records the databanks in the database.

Column name	Data type	Description
DATABANK_SYSID	INTEGER	System identifier for the databank.
DATABANK_SEQNO	INTEGER	The shadowing sequence number for the databank.
DATABANK_FILENO	INTEGER	The file number of the databank file.
DATABANK_NAME	NCHAR VARYING(128)	The name of the databank.
SHADOW_NAME	NCHAR VARYING(128)	The shadow name if the databank is a shadow.
DATABANK_FILENAME	NCHAR VARYING(256)	The path name for the databank file.
DATABANK_TYPE	VARCHAR(20)	Indicates type of transaction handling: LOG = transaction handling with logging READ ONLY = allows read operations only TEMPORARY = work databank (SQLDB) TRANSACTION = transaction handling without logging WORK = transaction handling not used
IS_MASTER	BOOLEAN	One of: FALSE = the databank is a shadow TRUE = the databank is a master.
IS_ONLINE	BOOLEAN	One of: FALSE = the databank is OFFLINE TRUE = the databank of ONLINE.

Column name	Data type	Description
LOBDIR_SYSID	INTEGER	System identifier for the large object directory in the databank.
LOBDATA_SYSID	INTEGER	System identifier for the large object table in the databank
MAXSIZE	BIGINT	Maximum size for databank file, kilo bytes
GOALSIZE	BIGINT	Ideal size for databank file, kilo bytes
MINSIZE	BIGINT	Minimum size for databank file, kilo bytes
IS_REMOVABLE	BOOLEAN	One of: FALSE = databank is not removable TRUE = databank is removable
BACKUPED	TIMESTAMP(2)	Last date for backup, currently always set to null
SEQTABLE_SYSID	INTEGER	System id for the sequence table. Null if no sequence in databank.

Primary key: DATABANK_SYSID, DATABANK_SEQNO, DATABANK_FILENO

Unique constraint: SHADOW_NAME, DATABANK_FILENO

Secondary index: DATABANK_NAME, DATABANK_SEQNO

SYSTEM.DIRECT_SUPERTYPES

Contains information about inheritance relation between user-defined types. (Currently not used.)

Column name	Data type	Description
UDT_SYSID	INTEGER	System identifier for the user-defined type.
UDT_SUPERTYPE_SYSID	INTEGER	System identifier for user-defined type used in inheritance.

Primary key: UDT_SYSID, UDT_SUPERTYPE_SYSID

Secondary index: UDT_SUPERTYPE_SYSID

SYSTEM.DOMAINS

Records the domains in the database.

Column name	Data type	Description
DOMAIN_SYSID	INTEGER	System identifier for the domain.
COLLATION_SYSID	INTEGER	System identifier for the collation for a domain associated with a character set.
CHARSET_SYSID	INTEGER	System identifier for the character set.
DATA_TYPE	VARCHAR(30)	Identifies the data type of the domain. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier for the internal data type.

Column name	Data type	Description
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	This defines the total number of significant digits to the right of the decimal point. For INTEGER and SMALLINT, this is 0. For CHARACTER, VARCHAR, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, it is the null value.
NUMERIC_PREC_RADIX	INTEGER	For NUMERIC data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.

Column name	Data type	Description
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and interval data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_LENGTH	INTEGER	Internal length in bytes of value.

Column name	Data type	Description
DOMAIN_DEFAULT	NCHAR VARYING(200)	This shows the default value for the domain. If the default value is a character string, the value shown is the string enclosed in single quotes. If the default value is a numeric literal, the value is shown in its original character representation without enclosing quotes. If the default value is a DATE, TIME or TIMESTAMP, the value shown is the appropriate keyword (e.g. DATE) followed by the literal representation of the value enclosed in single quotes, see *Mimer SQL Reference Manual* for a description of DATE, TIME and TIMESTAMP literals. If the default value is a pseudo-literal, the value shown is the appropriate keyword (e.g. CURRENT_DATE) without enclosing quotes. If the default value is the null value, the value shown is the keyword NULL without enclosing quotes. If the default value cannot be represented without truncation, then a zero-length string is stored. If no default value was specified then its value is the null value. The value of DOMAIN_DEFAULT is syntactically suitable for use in specifying default-value in a CREATE TABLE or ALTER TABLE statement.
ANY_CONSTRAINT	BOOLEAN	One of: FALSE = There are no constraints for the domain TRUE = There are constraints for the domain.
IS_LEVEL2_APPROVED	BOOLEAN	One of: FALSE = Tables using this domain can not be used with the level 2 interface TRUE = The use of this domain does not prohibit the table being used with the level 2 interface.

Primary key: DOMAIN_SYSID

SYSTEM.DOMAIN_CONSTRAINTS

Records the constraints defined for domains in the database.

Column name	Data type	Description
DOMAIN_SYSID	INTEGER	System identifier for the domain.
CONSTRAINT_SYSID	INTEGER	System identifier for the constraint.
IS_DEFERRABLE	BOOLEAN	One of: FALSE = the constraint is not deferrable. TRUE = the constraint is deferrable.
INITIALLY_DEFERRED	BOOLEAN	One of: FALSE = the constraint is not initially deferred TRUE = the constraint is initially deferred.

Primary key: DOMAIN_SYSID, CONSTRAINT_SYSID

Secondary index: CONSTRAINT_SYSID

SYSTEM.EXEC_STATEMENTS

Records precompiled statements in the database.

Column name	Data type	Description
STATEMENT_SYSID	INTEGER	System identifier for the precompiled statement.
STATEMENT_TYPE	INTEGER	Type of statement (internal value).
STATEMENT_SOU_LENGTH	INTEGER	Length of source text in characters.
STATEMENT_SOURCE	NCHAR VARYING(200)	Source text for the precompiled statement. If the length exceeds 200 characters, this value is null and the source text is stored in SOURCE_DEFINITION.
INLINE	BOOLEAN	One of: FALSE = the statement is not inline TRUE = the statement is inline.

Primary key: STATEMENT_SYSID

SYSTEM.FIPS_FEATURES

Lists details of all FIPS features.

Column name	Data type	Description
FEATURE_ID	SMALLINT	Identification number of the FIPS feature.
FEATURE_NAME	VARCHAR(50)	The name of the FIPS feature.
CLASSIFICATION	VARCHAR(12)	The conformance level of the feature. One of: TRANSITIONAL INTERMEDIATE FULL.
IS_SUPPORTED	BOOLEAN	One of: TRUE = Mimer SQL supports the feature FALSE = Mimer SQL does not support the feature.
IS_VERIFIED	BOOLEAN	One of: TRUE = Mimer SQL support for the feature has been tested and verified FALSE = Mimer SQL support for the feature has not been verified.
FEATURE_COMMENTS	VARCHAR(100)	Comments about the feature.

Primary key: FEATURE_ID

SYSTEM.FIPS_SIZING

Lists details about FIPS limits.

Column name	Data type	Description
SIZING_ID	SMALLINT	Identification number for limit.
DESCRIPTION	VARCHAR(50)	Description of limit.
ENTRY_VALUE	INTEGER	Limit for entry SQL.
INTERMEDIATE_VALUE	INTEGER	Limit for intermediate SQL.
VALUE_SUPPORTED	INTEGER	Limit for Mimer SQL.
SIZING_COMMENTS	VARCHAR(100)	Comments for limit.

Primary key: SIZING_ID

SYSTEM.HEURISTICS

Holds information about distributed transactions.

Column name	Data type	Description
COMMIT_ID	BINARY(128)	
HEURISTIC_OPCODE	INTEGER	
COMMIT_PROTOCOL	INTEGER	
FORMAT_ID	INTEGER	
GTRID_LENGTH	INTEGER	
BQUAL_LENGTH	INTEGER	

Primary key: COMMIT_ID

SYSTEM.KEY_COLUMN_USAGE

Records columns in an index or in a primary, unique or foreign key.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier for the table.
CONSTRAINT_SYSID	INTEGER	System identifier for the table constraint.
ORDINAL_POSITION	INTEGER	The ordinal position of the column in the table. The first column in the table is number 1.
COLUMN_ID	INTEGER	The id of the table column.
KEY_TYPE	VARCHAR(20)	One of: FOREIGN KEY INDEX INTERNAL KEY PRIMARY KEY UNIQUE.
IS_UNIQUE	BOOLEAN	One of: FALSE = the column may contain non-unique values TRUE = the column is constrained to contain only unique values.
IS_ASCENDING	BOOLEAN	One of: FALSE = the column is indexed in descending value order TRUE = the column is indexed in ascending value order.

Column name	Data type	Description
COLLATION_SYSID	INTEGER	System identifier for the collation used by the index.
INDEX_ALGORITHM	VARCHAR(20)	Algorithm used when creating index.
ATTRIBUTE_SYSID	INTEGER	System identifier for user-defined type if index defined on attribute in structured type.
ATTRIBUTE_NUMBER	INTEGER	Ordinal position for attribute within user defined structured type.
ATTRIBUTE_OFFSET	INTEGER	Offset for attribute within user-defined type.
INTERNAL_TYPE	INTEGER	Data type for index column.
SIZE	INTEGER	Precision of index column data type.
SCALE	INTEGER	Scale for index column data type.

Primary key: TABLE_SYSID, CONSTRAINT_SYSID, ORDINAL_POSITION

Unique constraint: CONSTRAINT_SYSID, COLUMN_ID

SYSTEM.LEVEL2_RESTRICT

Records restrictions for domains legal for use by Level 2.

Column name	Data type	Description
DOMAINID	INTEGER	System identifier for the domain with restrictions.
SEQNO	SMALLINT	Restriction order number.
LOGOP	CHAR(3)	Logical operator (AND, OR).
RELOP	CHAR(2)	Relational operator (EQ, …).
LENGTH	SMALLINT	Length of the value.
RESVALUE	VARCHAR(64)	Restriction value.

Primary key: DOMAINID, SEQNO

SYSTEM.LEVEL2_VIEWCOL

Records columns of views acceptable by DB level 2.

Column name	Data type	Description
VTABID	INTEGER	View table identifier.
VCOLNO	SMALLINT	View table column number.
BTABID	INTEGER	Base table identifier.
BCOLNO	SMALLINT	Base table column number.

Primary key: VTABID, VCOLNO

SYSTEM.LEVEL2_VIEWRES

Records restrictions for DB level 2.

Column name	Data type	Description
VTABID	INTEGER	View table identifier.
SEQNO	SMALLINT	Restriction order number.
LOGOP	CHAR(3)	Logical operator (AND, OR).
BTABID	INTEGER	Base table identifier.
BCOLNO	SMALLINT	Base table column number.
RELOP	CHAR(2)	Relational operator (EQ, …).
LENGTH	SMALLINT	Length of the value.
RESVALUE	VARCHAR(64)	Restriction value.

Primary key: VTABID, SEQNO

SYSTEM.LIBRARIES

Records external libraries.

Column name	Data type	Description
LIBRARY_SYSID	INTEGER	System identifier for external library.
LIBRARY_FILENAME	NCHAR VARYING(256)	Filename for external library.

Primary key: LIBRARY_SYSID

Secondary index: LIBRARY_FILENAME

SYSTEM.LOGINS

Records OS_USER logins for user idents.

Column name	Data type	Description
USER_LOGIN	NCHAR VARYING(128)	OS_USER name.
USER_SYSID	INTEGER	System identifier for the ident.

Primary key: USER_LOGIN, USER_SYSID

Secondary index: USER_SYSID

SYSTEM.MANYROWS

Dummy table with more than one row.

Column name	Data type	Description
C	SMALLINT	The values column.

Primary key: C

SYSTEM.MESSAGE

Records translations of message codes to message text.

Column name	Data type	Description
MODULEID	INTEGER	Identification number for Mimer SQL module to which error belongs.
MESSAGEID	INTEGER	Identification number for message
LANGUAGE	INTEGER	Language number for message 1 = English.
LINENO	INTEGER	Line number for message.
MESSAGE	VARCHAR(80)	Message text.

Primary key: MODULEID, MESSAGEID, LANGUAGE, LINENO

SYSTEM.METHOD_SPECIFICATION_PARAMETERS

Records parameters to method specifications.

Column name	Data type	Description
METHOD_SYSID	INTEGER	System identifier for the method specification.
ORDINAL_POSITION	INTEGER	The ordinal position of the parameter in the method specification. The first parameter is number 1.
PARAMETER_NAME	NCHAR VARYING(128)	The name of the parameter.
PARAMETER_MODE	CHAR(5)	One of: IN OUT INOUT.
IS_RESULT	BOOLEAN	One of TRUE = The parameter is a result parameter FALSE = The parameter is not a result parameter
UDT_SYSID	INTEGER	System identifier for the parameter's type, if it's a user-defined type.
COLLATION_SYSID	INTEGER	System identifier of the collation associated with the parameter.
CHARSET_SYSID	INTEGER	The system identifier for the character set.

Column name	Data type	Description
DATA_TYPE	VARCHAR(30)	The data type of the parameter. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
INTERVAL_TYPE	CHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier for the internal data type.

Column name	Data type	Description
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	This defines the total number of significant digits to the right of the decimal point. For INTEGER and SMALLINT, this is 0. For CHARACTER, VARCHAR, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, it is the null value.

Column name	Data type	Description
NUMERIC_PREC_RADIX	INTEGER	For numeric data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
AS_LOCATOR	BOOLEAN	One of: TRUE = declared as locator FALSE = not locator
INTERNAL_LENGTH	INTEGER	Internal length.
DOMAIN_SYSID	INTEGER	System identifier for domain, when a domain is used as the type for a parameter in a method specification.

Primary key: METHOD_SYSID, ORDINAL_POSITION

Unique constraint: METHOD_SYSID, PARAMETER_NAME

SYSTEM.METHOD_SPECIFICATIONS

Records method specifications.

Column name	Data type	Description
METHOD_SYSID	INTEGER	System identifier for the method specification.
UDT_SYSID	INTEGER	System identifier for user-defined type, to which the method is specified.
IS_STATIC	BOOLEAN	One of TRUE = Method specification is static FALSE = Method specification is not static
IS_OVERRIDING	BOOLEAN	One of TRUE = Method specification is overriding FALSE = Method specification is not overriding
IS_CONSTRUCTOR	BOOLEAN	One of TRUE = Method specification is a constructor method FALSE = Method specification is not a constructor method
IS_DETERMINISTIC	BOOLEAN	One of: FALSE = The routine is not deterministic. i.e. invoking the routine with the same input values is not guaranteed to return the same result TRUE = The routine is deterministic, i.e. when invoked with same input values it will always return the same result.
IS_NULL_CALL	BOOLEAN	One of: TRUE = method will be invoked when parameters are null FALSE = method will not be invoked when parameters are null and returns null. (currently not used)
SQL_DATA_ACCESS	VARCHAR(20)	One of: NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA.
METHOD_LANGUAGE	VARCHAR(20)	Language used for method. One of: SQL (currently not used)

Column name	Data type	Description
PARAMETER_STYLE	VARCHAR(20)	The parameter passing style of the routine if it is an external routine, otherwise the null value is shown.
RESULT_UDT_SYSID	INTEGER	System identifier for the result data type, if it's a user-defined type.
COLLATION_SYSID	INTEGER	System identifier of the collation associated with the result.
CHARSET_SYSID	INTEGER	The system identifier for the character set.
DATA_TYPE	VARCHAR(30)	The data type of the result. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
INTERVAL_TYPE	VARCHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier for the internal data type.

Column name	Data type	Description
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	This defines the total number of significant digits to the right of the decimal point. For INTEGER and SMALLINT, this is 0. For CHARACTER, VARCHAR, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, it is the null value.
NUMERIC_PREC_RADIX	INTEGER	For numeric data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.

Column name	Data type	Description
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
TOTAL_PARAMS	INTEGER	The number of parameters. (Described in *SYSTEM.METHOD_SPECIFICATION_PARAMETERS* on page *205*.)
AS_LOCATOR	BOOLEAN	One of: TRUE = declared as locator FALSE = not locator
INTERNAL_LENGTH	INTEGER	Internal length.
SELF_AS_RESULT	BOOLEAN	One of: TRUE = returns self FALSE = does not return self
DOMAIN_SYSID	INTEGER	System identifier for domain, when a domain is used as the return type for a method.

Primary key: METHOD_SYSID

Secondary index: UDT_SYSID

SYSTEM.MODULES

Records the SQL-server modules in the database.

Column name	Data type	Description
MODULE_SYSID	INTEGER	System identifier for the module.
DEF_CHARSET_SYSID	INTEGER	System identifier for the default character set for the module.
DEF_SCHEMA_SYSID	INTEGER	System identifier for the default schema for the module.
MODULE_LENGTH	INTEGER	Length of the module definition.
MODULE_DEFINITION	NCHAR VARYING(200)	Source text for module definition, if the length of the module definition exceeds 200 characters, this value is null and the source text is stored in SOURCE_DEFINITION.

Primary key: MODULE_SYSID

SYSTEM.NANO_DATABANKS

Contains information about Mimer Nano databanks. (Currently not used.)

SYSTEM.NANO_DESCRIPTORS

Contains information about Mimer Nano descriptors. (Currently not used.)

SYSTEM.NANO_OBJECTS

Contains information about Mimer Nano objects. (Currently not used.)

SYSTEM.NANO_ROUTINE_USE

Contains information about Mimer Nano routine usage. (Currently not used.)

SYSTEM.NANO_USERS

Contains information about Mimer Nano databanks. (Currently not used.)

SYSTEM.OBJECT_COLUMN_USE

Records columns referenced by other database objects.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier for the table.
COLUMN_ID	INTEGER	The id of the table column.
USED_BY_SYSID	INTEGER	The system identifier of the referencing object.
USED_BY_TYPE	VARCHAR(20)	One of: VIEW PROCEDURE FUNCTION METHOD CHECK TRIGGER USER DEFINED TYPE STATEMENT.
PRIVILEGE_TYPE	VARCHAR(20)	One of: INSERT DELETE SELECT UPDATE REFERENCES.

Primary key: TABLE_SYSID, COLUMN_ID, USED_BY_SYSID, USED_BY_TYPE, PRIVILEGE_TYPE

Secondary index: USED_BY_SYSID

SYSTEM.OBJECT_OBJECT_USE

Records objects referenced by other objects.

Column name	Data type	Description
OBJECT_SYSID	INTEGER	System identifier for the object used.
USED_BY_SYSID	INTEGER	The system identifier of the referencing object.
IS_STRONG	BOOLEAN	Determines if a dependent object will be dropped when dropping the object on which this object depends. One of: FALSE = the dependent object will be dropped even if drop restrict is specified TRUE = the object will only be dropped if cascade is specified.
OBJECT_TYPE	VARCHAR(20)	One of: BASE TABLE DATABANK INDEX VIEW PROCEDURE FUNCTION METHOD METHOD SPECIFICATION DOMAIN SEQUENCE CHARACTER SET COLLATION TRANSLATION TRIGGER USER DEFINED TYPE STATEMENT.

Column name	Data type	Description
USED_BY_TYPE	VARCHAR(20)	One of: BASE_TABLE VIEW PROCEDURE FUNCTION METHOD METHOD SPECIFICATION DOMAIN TRIGGER CHARACTER SET COLLATION CHECK INDEX STATEMENT.

Primary key: OBJECT_SYSID, USED_BY_SYSID

Secondary index: USED_BY_SYSID

SYSTEM.OBJECT_PROGRAMS

Contains information about predefined executable statements for table operations and routines.

Column name	Data type	Description
OBJECT_SYSID	INTEGER	System identifier for object.
OPERATION	VARCHAR(20)	Type of operation, one of: DELETE INSERT UPDATE '' *(empty string)*, routine invocation
STATEMENT_SYSID	INTEGER	System identifier for executable statement.

Primary key: OBJECT_SYSID, OPERATION

Secondary index: STATEMENT_SYSID

SYSTEM.OBJECTS

Records objects in the database.

Column name	Data type	Description
OBJECT_SYSID	INTEGER	System identifier for the database object.
OBJECT_TYPE	VARCHAR(20)	One of: BASE TABLE CHARACTER SET COLLATION CONSTRAINT CONSTRUCTOR METHOD DATABANK DOMAIN FUNCTION IDENT INDEX INSTANCE METHOD METHOD SPECIFICATION MODULE PROCEDURE SCHEMA SEQUENCE SHADOW STATEMENT STATIC METHOD SYNONYM TRANSLATION TRIGGER USER DEFINED TYPE VIEW.
OBJECT_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the object.
OBJECT_NAME	NCHAR VARYING(128)	The name of the object.
OBJECT_CREATED	TIMESTAMP(2)	The date and time the object was created.
OBJECT_ALTERED	TIMESTAMP(2)	The date and time the object was last altered.
IS_IMPLICIT	BOOLEAN	One of: TRUE = the object is created implicitly as the result of another CREATE statement FALSE = the object is created explicitly.

Column name	Data type	Description
SPECIFIC_SYSID	INTEGER	Specific id for overloaded routines.
COMPATIBILITY	INTEGER	Indicates which compatibility mode was set when the object was created.
LAST_DEPENDENCY_SYSID	INTEGER	Upper limit for scope when resolving object references in routines and triggers.

Primary key: OBJECT_SYSID

Unique constraint: OBJECT_TYPE, OBJECT_SCHEMA, OBJECT_NAME, SPECIFIC_SYSID

SYSTEM.ONEROW

Dummy table containing one row.

Column name	Data type	Description
M	CHAR(1)	Contains the value M.

Primary key: M

SYSTEM.PARAMETERS

Records parameters of routines in the database.

Column name	Data type	Description
ROUTINE_SYSID	INTEGER	System identifier for the function or procedure.
ORDINAL_POSITION	INTEGER	The ordinal position of the parameter in the routine. The first parameter in the routine is number 1.
IS_RETURN	BOOLEAN	For a type preserving method this column indicates whether the parameter is type preserving or not. One of TRUE FALSE
PARAMETER_MODE	CHAR(5)	One of: IN OUT INOUT.

Column name	Data type	Description
PARAMETER_NAME	NCHAR VARYING(128)	The name of the parameter.
DOMAIN_SYSID	INTEGER	System identifier of the domain that defines the data type of the parameter.
COLLATION_SYSID	INTEGER	System identifier of the collation associated with the parameter.
CHARSET_SYSID	INTEGER	The system identifier for the character set.
UDT_SYSID	INTEGER	System identifier for a user-defined type.
DATA_TYPE	CHAR(30)	The data type of the parameter. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
INTERVAL_TYPE	CHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier for the internal data type.

Column name	Data type	Description
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	This defines the total number of significant digits to the right of the decimal point. For INTEGER and SMALLINT, this is 0. For CHARACTER, VARCHAR, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, it is the null value.
NUMERIC_PREC_RADIX	INTEGER	For numeric data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.

Column name	Data type	Description
PARAMETER_NAME	NCHAR VARYING(128)	The name of the parameter.
DOMAIN_SYSID	INTEGER	System identifier of the domain that defines the data type of the parameter.
COLLATION_SYSID	INTEGER	System identifier of the collation associated with the parameter.
CHARSET_SYSID	INTEGER	The system identifier for the character set.
UDT_SYSID	INTEGER	System identifier for a user-defined type.
DATA_TYPE	CHAR(30)	The data type of the parameter. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
INTERVAL_TYPE	CHAR(30)	For INTERVAL data types, this is a character string specifying the interval qualifier for the named interval data type, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier for the internal data type.

Column name	Data type	Description
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	This defines the total number of significant digits to the right of the decimal point. For INTEGER and SMALLINT, this is 0. For CHARACTER, VARCHAR, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, it is the null value.
NUMERIC_PREC_RADIX	INTEGER	For numeric data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.

Column name	Data type	Description
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and INTERVAL data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
AS_LOCATOR	BOOLEAN	One of: TRUE = declared as locator FALSE = not locator
INTERNAL_LENGTH	INTEGER	Internal length.
IS_RESULT	BOOLEAN	One of: FALSE = This parameter is not part of the result clause for a result set procedure TRUE = This parameter is part of the result set.

Primary key: ROUTINE_SYSID, ORDINAL_POSITION, IS_RETURN

SYSTEM.REFER_CONSTRAINTS

Records referential constraints in the database.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier for the table.
CONSTRAINT_SYSID	INTEGER	System identifier for the referential constraint.
UNIQUE_TABLE_SYSID	INTEGER	System identifier for the table that is referenced in the foreign key.
UNIQUE_CONST_SYSID	INTEGER	System identifier for the constraint that is (implicitly) referenced in the foreign key.

Column name	Data type	Description
IS_CONSISTENT	BOOLEAN	Indicates if the foreign key constraint is consistent. One of: TRUE FALSE A foreign key constraint will be marked as inconsistent if the system databank transdb is dropped or if the databank in which the referenced or referencing table is located is set to work option.
MATCH_OPTION	VARCHAR(20)	One of: NONE PARTIAL FULL.
UPDATE_RULE	VARCHAR(20)	One of: CASCADE SET NULL SET DEFAULT NO ACTION RESTRICT.
DELETE_RULE	VARCHAR(20)	One of: CASCADE SET NULL SET DEFAULT NO ACTION RESTRICT.

Primary key: TABLE_SYSID, CONSTRAINT_SYSID

Secondary index: UNIQUE_TABLE_SYSID, UNIQUE_CONST_SYSID

Secondary index: CONSTRAINT_SYSID

Secondary index: UNIQUE_CONST_SYSID

SYSTEM.ROUTINES

Records procedures and user-defined functions in the database.

Column name	Data type	Description
ROUTINE_SYSID	INTEGER	System identifier of the function or procedure.
MODULE_SYSID	INTEGER	System identifier of the module to which the routine belongs.
ROUTINE_TYPE	VARCHAR(20)	One of: PROCEDURE FUNCTION.
SQL_DATA_ACCESS	VARCHAR(20)	One of: NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA.
ROUTINE_OFFSET	INTEGER	Offset for routine within module definition. This value is zero if the routine ins not defined in a module.
ROUTINE_LENGTH	INTEGER	Length of routine definition
TOTAL_PARAMS	INTEGER	The total number of routine parameters.
RESULT_PARAMS	INTEGER	The number of result parameters.
INPUT_PARAMS	INTEGER	The number of input parameters.
OUTPUT_PARAMS	INTEGER	The number of output parameters.
ROUTINE_DEFINITION	NCHAR VARYING(200)	Routine definition. If the length of the routine definition exceeds 200 characters, this value is null and the source text is stored in SOURCE_DEFINITION.

Column name	Data type	Description
IS_DETERMINISTIC	BOOLEAN	One of: FALSE = The routine is not deterministic. i.e. invoking the routine with the same input values is not guaranteed to return the same result TRUE = The routine is deterministic, i.e. when invoked with same input values it will always return the same result.
ROUTINE_BODY	VARCHAR(20)	One of: SQL = The routine is an SQL routine EXTERNAL = The routine is external
EXTERNAL_NAME	NCHAR VARYING(128)	The name of the program that implements the routine if it is an external routine otherwise null.
EXTERNAL_LANGUAGE	VARCHAR(20)	The language for the routine if it is external otherwise null.
PARAMETER_STYLE	VARCHAR(20)	The parameter passing style for the routine if it is an external routine, otherwise null.
SCHEMA_LEVEL_ROU	BOOLEAN	One of: FALSE = The routine is defined within a module TRUE = The routine is not defined in a module.
MX_DYN_RESULT_SETS	INTEGER	The maximal number of result sets that the routine may return.
UDT_SYSID	INTEGER	System identifier for user-defined type, if routine is a method.
IS_USER_DEFINED_CAST	BOOLEAN	(currently not used)
IS_IMPLICITLY_INVOCABLE	BOOLEAN	(currently not used)
MS_SYSID	INTEGER	(currently not used)

Column name	Data type	Description
IS_NULL_CALL	BOOLEAN	One of: TRUE = routine will be called if all parameters are null FALSE = routine returns null if all parameters are null
OVERRIDING_SYSID	INTEGER	System identifier for original method specification.
ROUTINE_HEADER_OFFSET	INTEGER	Offset for routine header in source definition.
ROUTINE_HEADER_LENGTH	INTEGER	Length of routine header.
ROUTINE_RETURNS_OFFSET	INTEGER	Offset for returns clause in source definition.
ROUTINE_RETURNS_LENGTH	INTEGER	Length of return clause.
ROUTINE_SPECIFIC_OFFSET	INTEGER	Offset for specific clause in source definition.
ROUTINE_SPECIFIC_LENGTH	INTEGER	Length of specific clause.
ROUTINE_EXTERNAL_OFFSET	INTEGER	Offset for external clause in source definition.
ROUTINE_EXTERNAL_LENGTH	INTEGER	Length of external clause.
ROUTINE_BODY_OFFSET	INTEGER	Offset for routine body in source definition.
ROUTINE_BODY_LENGTH	INTEGER	Length of routine body.
LIBRARY_SYSID	INTEGER	System identifier for external library.
ROUTINES_IS_INVOKED_MANY	BOOLEAN	The column indicates whether a function or method should be invoked multiple times or only once within a statement.

Primary key: ROUTINE_SYSID

Secondary index: LIBRARY_SYSID

Secondary index: MODULE_SYSID

Secondary index: OVERRIDING_SYSID

Unique constraint: MS_SYSID

SYSTEM.SCHEMATA

Records schemas in the database.

Column name	Data type	Description
SCHEMA_SYSID	INTEGER	System identifier of the schema.
DEF_CHARSET_SYSID	INTEGER	System identifier of the default character set.

Primary key: SCHEMA_SYSID

SYSTEM.SEQUENCE_VALUE_TABLE

Stores information about sequences.

Column name	Data type	Description
SEQ_SYSID	INTEGER	System identifier of the sequence.
CURRENT_VALUE	BIGINT	Current value for sequence.
CYCLES	BIGINT	Number of times the sequence has reached its maximum value.
IS_EXHAUSTED	INTEGER	Indicates whether a unique sequences has reached its maximum value. One of: 1 = sequence has reached its maximum value 0 = sequence has not reached its maximum value

Primary key: SEQ_SYSID

SYSTEM.SEQUENCES

Records sequences in the database.

Column name	Data type	Description
SEQUENCE_SYSID	INTEGER	System identifier of the sequence.
DATA_TYPE	VARCHAR(30)	One of: SMALLINT INTEGER BIGINT.
INTERNAL_TYPE	INTEGER	System identifier of internal data type.
NUMERIC_PRECISION	INTEGER	Precision for data type.
MAXIMUM_VALUE	BIGINT	The maximum value of the sequence.
MINIMUM_VALUE	BIGINT	The minimum value of the sequence.
INCREMENT	BIGINT	The increment value for sequence.
CYCLE_OPTION	BOOLEAN	One of: TRUE = the sequence will restart when all values are exhausted FALSE = the sequence will not restart when all values are exhausted.
START_VALUE	BIGINT	Start value for sequence when first used.
CURRENT_VALUE	BIGINT	Start value at system start up.
IS_EXHAUSTED	BOOLEAN	One of: TRUE = the series of values defined by the sequence is exhausted FALSE = the series of values defined by the sequence is not exhausted.
CYCLES	BIGINT	Number of times the maximum value of the sequence is exceeded.
FLUSH_RATE	BIGINT	Number of sequence value allocations between saving sequence data to disk.
DATABANK_SYSID	INTEGER	Databank location.
INTERNAL_LENGTH	INTEGER	Internal length.

Primary key: SEQUENCE_SYSID

SYSTEM.SERVER_INFO

Records attributes of the current database system or server.

Column name	Data type	Description
SERVER_ATTRIBUTE	VARCHAR(100)	One of: AUTOUPGRADE_ENABLED CATALOG_NAME COLLATION_SEQ CURRENT_COLLATION_ID IDENTIFIER_LENGTH INTERVAL_FRAC_PREC INTERVAL_LEAD_PREC ROW_LENGTH TIME_PREC TIMESTAMP_PREC TXN_ISOLATION USERID_LENGTH CATALOG_VERSION_CREATED CATALOG_VERSION_CURRENT.
ATTRIBUTE_VALUE	VARCHAR(100)	The value for the attribute.

Primary key: SERVER_ATTRIBUTE

SYSTEM.SEVERITY

Records severity levels and optional module for error codes.

Column name	Data type	Description
MODULEID	INTEGER	Identification number for the Mimer SQL module to which the message belongs.
MESSAGEID	INTEGER	Identification number for message.
SEVERITY	INTEGER	Severity for message: 1 = Message 2 = Warning 3 = Error 4 = Fatal error 5 = Internal error 6 = Code.
MODULE	VARCHAR(20)	Name of Mimer SQL module to which the message belongs.
SQLSTATE_VALUE	CHAR(5)	Corresponding SQLSTATE value for message.
CLASS_ORIGIN	VARCHAR(20)	Class origin for the SQLSTATE value.
SUBCLASS_ORIGIN	VARCHAR(20)	Sub class origin for the SQLSTATE value.

Primary key: MODULEID, MESSAGEID

SYSTEM.SOURCE_DEFINITION

Records source definitions for defaults, check constraints, views, modules, procedures, functions, triggers and statements.

Column name	Data type	Description
SOURCE_SYSID	INTEGER	System identifier of the source definition.
COLUMN_ID	INTEGER	Column id if the source types is a default value for a column. Null otherwise.
SOURCE_TYPE	VARCHAR(20)	One of: DEFAULT CHECK VIEW ROUTINE STATEMENT.
SEQUENCE_NO	INTEGER	Sequence number in source definition.

Column name	Data type	Description
SOURCE_LENGTH	INTEGER	The length of the stored source text.
SOURCE_DEFINITION	NCHAR VARYING(400)	The stored source text.

Primary key: SOURCE_SYSID, COLUMN_ID, SOURCE_TYPE, SEQUENCE_NO

SYSTEM.SPECIFIC_NAMES

Records specific names of the routines in the database.

Column name	Data type	Description
ROUTINE_SYSID	INTEGER	System identifier of the routine.
SPECIFIC_SCHEMA	NCHAR VARYING(128)	The name of the schema containing the routine.
SPECIFIC_NAME	NCHAR VARYING(128)	The specific name for the routine.
UDT_SYSID	INTEGER	System identifier for user-defined type if routine is a method.

Primary key: ROUTINE_SYSID

Unique constraint: SPECIFIC_SCHEMA, SPECIFIC_NAME

SYSTEM.SQL_CONFORMANCE

The SQL_CONFORMANCE base table has one row for each conformance element identified by ISO/IEC 9075.

Column name	Data type	Description
CONFORMANCE_ID	VARCHAR(20)	Identification of the conformance element described.
CONFORMANCE_SUB_ID	VARCHAR(20)	If the conformance element is a subfeature then it's identification, otherwise a single space.
CONFORMANCE_TYPE	VARCHAR(20)	Type of conformance element. FEATURE, SUBFEATURE, PACKAGE, or PART.
CONFORMANCE_IS_CORE_SQL	BOOLEAN	TRUE if the conformance element belongs to Core SQL, otherwise FALSE.

Column name	Data type	Description
CONFORMANCE_NAME	VARCHAR(100)	Short description of the conformance element.
CONFORMANCE_SUB_NAME	VARCHAR(130)	Short description of a subfeature, otherwise a single space.
CONFORMANCE_IS_SUPPORTED	BOOLEAN	TRUE if fully supported by Mimer SQL, otherwise FALSE.
CONFORMANCE_IS_VERIFIED_BY	BOOLEAN	Should identify conformance test used to verify the conformance (always null).
CONFORMANCE_REMARKS	VARCHAR(100)	Comments pertinent to the conformance element.

Primary key: CONFORMANCE_ID, CONFORMANCE_SUB_ID

SYSTEM.SQL_LANGUAGES

Records the SQL standards and SQL dialects supported.

Column name	Data type	Description
ORDINAL_NO	INTEGER	Ordinal number.
SOURCE	VARCHAR(100)	Body that has defined standard.
SOURCE_YEAR	VARCHAR(100)	Which year the standard was approved.
CONFORMANCE	VARCHAR(100)	Level of conformance within standard.
INTEGRITY	VARCHAR(100)	If the supported standard is ISO 9075 the value YES in this column means that the integrity enhancement feature of this standard is supported, otherwise the value of this column is null.
IMPLEMENTATION	VARCHAR(100)	Implementation style, always null in Mimer SQL.
BINDING_STYLE	VARCHAR(100)	One of: DIRECT = Support for interactive (ad hoc) access to the database EMBEDDED = Support for access through an application programming interface.
PROGRAMMING_LANG	VARCHAR(100)	Name of programming languages that are supported if the binding style is EMBEDDED.

Primary key: ORDINAL_NO

SYSTEM.STATEMENT_DESCRIPTORS

Records compiled code for precompiled statements.

Column name	Data type	Description
STATEMENT_SYSID	INTEGER	System identifier for the precompiled statement.
DESC_TYPE	INTEGER	Internal. Naming or section descriptor type.
DESC_MODE	INTEGER	Mode, one of: 1 = no scroll 2 = scroll
DESC_SEQNO	INTEGER	Sequence number for compiled code.
DESC_VERSION	INTEGER	Compiler version used.
DESC_CODE_LENGTH	INTEGER	Length of compiled code.
DESC_CODE	BINARY(1000)	Compiled code.

Primary key: STATEMENT_SYSID, DESC_TYPE, DESC_MODE, DESC_SEQNO

SYSTEM.STATEMENT_ROUTINE_USE

Records compiled routines used by precompiled statements.

Column name	Data type	Description
STATEMENT_SYSID	INTEGER	System identifier for the precompiled statement.
USE_SEQNO	INTEGER	Sequence number.
USE_LEVEL	INTEGER	Call level for routine.
ROUTINE_SYSID	INTEGER	System identifier for called routines.

Primary key: STATEMENT_SYSID, USE_SEQNO

Secondary index: ROUTINE_SYSID

SYSTEM.SYNONYMS

Records synonyms and shadows in the database.

Column name	Data type	Description
SYNONYM_SYSID	INTEGER	System identifier of the synonym.
ORDINARY_SYSID	INTEGER	System identifier of the original object.

Primary key: SYNONYM_SYSID

Secondary index: ORDINARY_SYSID

SYSTEM.TABLES

Records tables and views in the database.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier of the table or view.
TABLE_TYPE	VARCHAR(20)	One of: BASE TABLE VIEW.
TABLE_NOCOLS	INTEGER	Number of columns in table.
TABLE_RECLEN	INTEGER	Record length for table.
TABLE_CARD	BIGINT	Number of records in table.
STATISTIC_GATHERED	TIMESTAMP(2)	Date and time when statistics for the table or view were lasted updated.
IS_LEVEL2_APPROVED	BOOLEAN	One of: TRUE = The table can be used with level 2 DB interface. FALSE = The table cannot be used with level 2 DB interface.
IS_PERSISTENT	BOOLEAN	One of: FALSE = The table is a temporary table TRUE = The table is not temporary.
DATABANK_SYSID	INTEGER	System identifier of the databank in which the table is stored.
LATEST_ROOTID	INTEGER	Identifier for the table used in the rootpage of the databank file.
VARIABLE	INTEGER	Number of length field entries.
IS_VARIABLE	BOOLEAN	One of: TRUE = The table has variable format FALSE = The table has fixed format

Column name	Data type	Description
COMMIT_ACTION	VARCHAR(20)	Indicates what happens with records in a temporary table at commit. One of: DELETE PRESERVE The column will be null if the table is not temporary.

Primary key: TABLE_SYSID

Secondary index: DATABANK_SYSID

SYSTEM.TABLE_CONSTRAINTS

Records table constraints in the database.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier of the table.
CONSTRAINT_SYSID	INTEGER	System identifier of the constraint.
CONSTRAINT_TYPE	VARCHAR(20)	One of: PRIMARY KEY UNIQUE FOREIGN KEY INDEX INDEX UNIQUE INTERNAL KEY CHECK.
CONSTRAINT_KEYCOLS	INTEGER	Number of key columns in the constraint.
CONSTRAINT_KEYLEN	INTEGER	Record length for key columns in the constraint.
CONSTRAINT_RECCOLS	INTEGER	Number of columns in constraint.
CONSTRAINT_RECLEN	INTEGER	Record length for constraint.
CONSTRAINT_CARD	BIGINT	Number of records in constraint.
DATABANK_SYSID	INTEGER	System identifier of the databank in which the table is stored.
IS_CONSISTENT	BOOLEAN	One of: FALSE = The index may be inconsistent and should not be used for index lookup only TRUE = The index may be used for index lookup only.

Column name	Data type	Description
IS_DEFERRABLE	BOOLEAN	One of: FALSE = The constraint is not deferrable TRUE = The constraint is deferrable.
INITIALLY_DEFERRED	BOOLEAN	One of: FALSE = The constraint is immediate TRUE = The constraint is not immediate.
VARIABLE	INTEGER	Number of length field entries.
LATEST_ROOTID	INTEGER	The latest value for the internal identifier for a constraint or index. The internal identifier can change if the table on which the constraint or index is defined is altered.
ON_CONFLICT	VARCHAR(20)	Possible behavior when a constraint is violated. Currently this value is always ABORT.

Primary key: TABLE_SYSID, CONSTRAINT_SYSID

Unique constraint: CONSTRAINT_SYSID

SYSTEM.TABLE_PRIVILEGES

Records instances of privileges granted on a table.

Column name	Data type	Description
TABLE_SYSID	INTEGER	System identifier of the table.
PRIVILEGE_TYPE	VARCHAR(20)	One of: DELETE INSERT LOAD REFERENCES SELECT UPDATE TRIGGER.
GRANTEE_SYSID	INTEGER	The sysid of the ident to whom the table privilege was granted.
GRANTOR_SYSID	INTEGER	The sysid of the ident granting the table privilege.
IS_GRANTABLE	BOOLEAN	One of: TRUE = WITH GRANT OPTION is held with the privilege FALSE = WITH GRANT OPTION is not held with the privilege.

Column name	Data type	Description
IS_INSTEAD_OF	BOOLEAN	One of: FALSE = The privilege was granted explicitly TRUE = The privilege was granted implicitly when an instead of trigger was created.
IS_ON_TABLE	BOOLEAN	One of: FALSE = The privilege was granted on individual columns TRUE = The privilege was granted on the complete table.
ALL_COLUMNS	BOOLEAN	One of: TRUE = the privilege was granted on the table and therefore applies to all table columns, including new ones added FALSE = the privilege only applies to the table columns explicitly specified when the privilege was granted.

Primary key: TABLE_SYSID, PRIVILEGE_TYPE, GRANTEE_SYSID, GRANTOR_SYSID

Secondary index: GRANTEE_SYSID

Secondary index: GRANTOR_SYSID

SYSTEM.TABLE_TYPES

Records the types of table supported.

Column name	Data type	Description
TABLE_TYPE	VARCHAR(20)	One of: SYNONYM SYSTEM TABLE TABLE VIEW.

Primary key: TABLE_TYPE

SYSTEM.TRANSLATIONS

Records character translations in the database.

Column name	Data type	Description
TRANSLATION_SYSID	INTEGER	System identifier of the character set translation.
SRC_CHARSET_SYSID	INTEGER	System identifier of the source character set for the translation.
TGT_CHARSET_SYSID	INTEGER	System identifier of the target character set for the translation.

Primary key: TRANSLATION_SYSID

SYSTEM.TRIGGERED_COLUMNS

Records table columns explicitly specified in an UPDATE TRIGGER event.

Column name	Data type	Description
TRIGGER_SYSID	INTEGER	System identifier of the trigger.
EVENT_TABLE_SYSID	INTEGER	System identifier of the table on which the trigger is defined.
EVENT_COLUMN_ID	INTEGER	The id of the column in which updates will cause the trigger to execute.

Primary key: TRIGGER_SYSID, EVENT_TABLE_SYSID, EVENT_COLUMN_ID

SYSTEM.TRIGGERS

Records triggers in the database.

Column name	Data type	Description
TRIGGER_SYSID	INTEGER	System identifier of the trigger.
EVENT_TABLE_SYSID	INTEGER	System identifier of the table on which the trigger is defined.
EVENT_MANIPULATION	VARCHAR(20)	One of: DELETE INSERT UPDATE.

Column name	Data type	Description
ACTION_ORDER	INTEGER	Ordinal number for trigger execution. This number will define the execution order of triggers on the same table and with the same value for EVENT_MANIPULATION, CONDITION_TIMING and ACTION_ORIENTATION. The trigger with 1 in this column will be executed first, followed by the trigger with 2 etc.
ACTION_CONDITION	NCHAR VARYING(200)	The text of the search condition of the trigger action WHEN clause.
ACTION_STATEMENT	NCHAR VARYING(200)	The character representation of the body of the trigger. If the length of the text exceeds 200 characters, this value is null and the source text is stored in SOURCE_DEFINITION.
ACTION_ORIENTATION	VARCHAR(20)	One of: ROW STATEMENT.
CONDITION_TIMING	VARCHAR(20)	One of: BEFORE AFTER INSTEAD OF.
COND_REF_NEW_TABLE	NCHAR VARYING(128)	Name of new table/row alias.
COND_REF_OLD_TABLE	NCHAR VARYING(128)	Name of old table/row alias.
REFERENCE_NEW	BOOLEAN	One of: FALSE = The trigger has no new table/row alias TRUE = The trigger has new table/row alias.
REFERENCE_OLD	BOOLEAN	One of: FALSE = The trigger has no new table/row alias TRUE = The trigger has new table/row alias.

Column name	Data type	Description
COLS_IS_IMPLICIT	BOOLEAN	One of: FALSE = The trigger is invoked on update on any column (if the event is update) TRUE = The trigger will only be invoked if a column in the for update of list is updated.
REF_SYSID	INTEGER	System identifier for foreign key constraint, if the trigger is defined for checking of a delete rule.

Primary key: TRIGGER_SYSID

Secondary index: EVENT_TABLE_SYSID

SYSTEM.TYPE_INFO

Records information about data types supported.

Column name	Data type	Description
INTERNAL_TYPE	INTEGER	System identifier of the internal data type.
DATA_TYPE	SMALLINT	Identification number for data type.

Column name	Data type	Description
TYPE_NAME	VARCHAR(30)	Name of data type. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.
COLUMN_SIZE	INTEGER	Length of data type name.
LITERAL_PREFIX	VARCHAR(30)	Optional prefix for a literal of this type.
LITERAL_SUFFIX	VARCHAR(30)	Optional suffix for a literal of this type.
CREATE_PARAMS	VARCHAR(30)	A description of how to specify length attributes for the data type.
NULLABLE	SMALLINT	One of: 1 = A not null constraint can be used with this type in a domain or column definition 0 = A not null constraint can not be used with this type in a domain or column definition.
CASE_SENSITIVE	SMALLINT	One of: 1 = The data type name is case sensitive 0 = The data type name is not case sensitive.

Column name	Data type	Description
SEARCHABLE	SMALLINT	One of: 0 = The data type can not be used with any comparison operator 1 = The data type can only be used with the LIKE operator 2 = The data type can be used with any comparison operator except like 3 = The data type can be used with any comparison operator.
UNSIGNED_ATTRIBUTE	SMALLINT	One of: 1 = The data type is unsigned 0 = The data type is signed The value is null if the data type is non-numeric.
FIXED_PREC_SCALE	SMALLINT	One of: 1 = The data type has a determined precision and scale 0 = The data type does not have a determined precision and scale.
AUTO_UNIQUE_VALUE	SMALLINT	One of: 1 = The data type will generate unique values 0 = The data type will not generate unique values.
LOCAL_TYPE_NAME	VARCHAR(30)	Local name for data type.
MINIMUM_SCALE	SMALLINT	Minimum value for scale for a numeric data type.
MAXIMUM_SCALE	SMALLINT	Maximum value for scale for a numeric data type.
SQL_DATA_TYPE	SMALLINT	A numeric value representing the data type.
SQL_DATETIME_SUB	SMALLINT	Subtype for an interval data type.
NUM_PREC_RADIX	SMALLINT	Radix for numeric data type.
INTERVAL_PRECISION	SMALLINT	Precision for an interval data type.
PRECISION	INTEGER	Precision for data type.
INTERNAL_LEN_DIFF	SMALLINT	Internal length difference.
INTERNAL_TYPENAME	VARCHAR(30)	Internal type name.
INTERNAL_VERSION	SMALLINT	Internal version number.
INTERNAL_TYPEORDER	SMALLINT	Internal type ordering number.

Primary key: INTERNAL_TYPE, DATA_TYPE

SYSTEM.USAGE_PRIVILEGES

Records instances of privileges which grant the right to use a database object.

Column name	Data type	Description
OBJECT_SYSID	INTEGER	System identifier of the database object.
OBJECT_PRIVILEGE	VARCHAR(20)	One of: BACKUP CHARACTER SET COLLATION DATABANK DOMAIN FUNCTION IDENT METHOD PROGRAM SCHEMA SEQUENCE_ON_DATABANK SHADOW STATEMENT STATISTICS TABLE USER DEFINED TYPE
GRANTEE_SYSID	INTEGER	The id of the ident to whom the usage privilege was granted.
GRANTOR_SYSID	INTEGER	The id of the ident granting the usage privilege.
IS_GRANTABLE	BOOLEAN	One of: TRUE = WITH GRANT OPTION is held with the privilege FALSE = WITH GRANT OPTION is not held with the privilege.
IS_IMPLICIT	BOOLEAN	One of: TRUE = The privilege has been granted implicitly as the result of another grant statement FALSE = The privilege has been granted implicitly.

Primary key: OBJECT_SYSID, OBJECT_PRIVILEGE, GRANTEE_SYSID, GRANTOR_SYSID

Secondary index: GRANTEE_SYSID

Secondary index: GRANTOR_SYSID

SYSTEM.USER_DEF_TYPES

Records user-defined types in the database.

Column name	Data type	Description
UDT_SYSID	INTEGER	System identifier of the user-defined type.
CATEGORY_TYPE	VARCHAR(20)	One of: STRUCTURED DISTINCT.
COLLATION_SYSID	INTEGER	System identifier for the collation used by the column.
CHARSET_SYSID	INTEGER	System identifier for the character set used by the column.
DATA_TYPE	VARCHAR(30)	Identifies the data type of the column. Can be one of the following: BIGINT BINARY BINARY VARYING BINARY LARGE OBJECT BOOLEAN CHARACTER CHARACTER VARYING CHARACTER LARGE OBJECT NATIONAL CHARACTER NATIONAL CHARACTER VARYING NATIONAL CHAR LARGE OBJECT DATE DECIMAL DOUBLE PRECISION FLOAT Float(p) INTEGER Integer(p) INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP USER-DEFINED.

Column name	Data type	Description
INTERVAL_TYPE	VARCHAR(30)	Identifies the interval type of the column
CHAR_MAX_LENGTH	BIGINT	For CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in characters or bytes as appropriate. For all other data types it is the null value.
CHAR_OCTET_LENGTH	BIGINT	For a CHARACTER, BINARY, CHARACTER LARGE OBJECT and BINARY LARGE OBJECT data types, this shows the maximum length in octets. For all other data types it is the null value. (For single octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUMERIC_PRECISION	INTEGER	For NUMERIC data types, this shows the total number of decimal digits allowed in the column. For all other data types it is the null value. NUMERIC_PREC_RADIX indicates the units of measurement.
NUMERIC_SCALE	INTEGER	For NUMERIC data types, this shows the total number of significant digits to the right of the decimal point. For INTEGER values this is 0. For all other types, it is the null value.

Column name	Data type	Description
NUMERIC_PREC_RADIX	INTEGER	For NUMERIC data types, the value 10 is shown because NUMERIC_PRECISION specifies a number of decimal digits. For all other data types it is the null value. NUMERIC_PRECISION and NUMERIC_PREC_RADIX can be combined to calculate the maximum number that the column can hold.
DATETIME_PRECISION	INTEGER	For DATE, TIME, TIMESTAMP and interval data types, this column contains the number of digits of precision for the fractional seconds component. For other data types it is the null value.
INTERVAL_PRECISION	INTEGER	For INTERVAL data types, this is the number of significant digits for the interval leading precision, see the *Mimer SQL Reference Manual*. For other data types it is the null value.
INTERNAL_TYPE	INTEGER	System identifier of the internal data type.
INTERNAL_LENGTH	INTEGER	Internal length in bytes of value.
MAX_INTERNAL_LENGTH	INTEGER	Length of the type's longest subtype.
IS_INSTANTIABLE	BOOLEAN	One of: TRUE = the structured user-defined type can be instantiated FALSE = the structured user-defined type cannot be instantiated.

Column name	Data type	Description
IS_FINAL	BOOLEAN	One of: TRUE = the structured user-defined type cannot be used as a supertype FALSE = the structured user-defined type can be used as a supertype.
ORDERING_FORM	VARCHAR(20)	One of: NONE, comparison not allowed for instances of the structured user-defined type EQUALS, only equality comparisons allowed for instances of the structured user-defined type FULL, all possible comparisons allowed for instances of the structured user-defined type.
ORDERING_CATEGORY	VARCHAR(20)	One of: RELATIVE, comparisons are done with a relative ordering function STATE, comparisons are done with a state ordering function MAP, comparisons are done with a map ordering function
ORDERING_ROUTINE_SYSID	INTEGER	System identifier for ordering function.
ORDERING_SYSID	INTEGER	System identifier for ordering.
NO_OF_ATTRIBUTES	INTEGER	Number of attributes in a structured user-defined type, 0 if user-defined type is distinct.
CONSTRUCTOR_FUNCTION	INTEGER	System identifier for constructor function, 0 if user-defined type is distinct.
OBSERVER_METHOD	INTEGER	System identifier for observer method, 0 if user-defined type is structured.

Column name	Data type	Description
TOTAL_DEFAULT_LENGTH	INTEGER	Total length in bytes for all default values for attributes in a structured user-defined type.
TOTAL_ATTRIBUTE_NAME_LENGTH	INTEGER	Total length for all attribute names in a structured user-defined type.
AS_LOCATOR	BOOLEAN	For a distinct type this indicates whether the base data type used in the type definition is declared AS LOCATOR or not. One of: TRUE FALSE
NO_OF_SUPERTYPES	INTEGER	Number of supertypes.
NO_OF_SUBTYPES	INTEGER	Number of subtypes.
NO_OF_INHERITED	INTEGER	Number of inherited.

Primary key: UDT_SYSID

Secondary index: ORDERING_SYSID

SYSTEM.USERS

Records idents (GROUP, PROGRAM or USER) in the database.

Column name	Data type	Description
USER_NAME	NCHAR VARYING(128)	The ident name.
USER_SYSID	INTEGER	System identifier for the ident.
USER_TYPE	VARCHAR(20)	One of: USER OS_USER PROGRAM GROUP.
USER_PASSWORD_MINIMUM_LENGTH	INTEGER	The minimum length for a password.

Primary key: USER_NAME

Unique constraint: USER_SYSID

SYSTEM.VIEWS

Records views in the database.

Column name	Data type	Description
VIEW_SYSID	INTEGER	System identifier of the view.
CHECK_OPTION	VARCHAR(20)	One of: CASCADED LOCAL NONE.
IS_UPDATABLE	BOOLEAN	One of: FALSE = The view can not be updated TRUE = The view can be updated.
IS_INSTEAD_OF	BOOLEAN	One of: FALSE = The view can be updated per se TRUE = The view can be updated due to the existence of an instead of trigger.
VIEW_DEFINITION	NCHAR VARYING(200)	The text of the view definition. If the length of the text exceeds 200 characters, this value is null and the text is stored in SOURCE_DEFINITION.

Primary key: VIEW_SYSID

Appendix E

System Limits

LIMIT	Value
Database cache	327 TB [a]
Databank file size	4096 PB

a. 2147480000 * (4k + 32k +128k blocks)

Appendix F

Deprecated Features

This chapter discusses features and functionality that have been deprecated.

Export/Import

The Export/Import functionality in UTIL has been deprecated. It has been replaced with Mimer SQL LOAD/UNLOAD, see *Chapter 8, Loading and Unloading Data and Definitions*.

Load/Unload

The Load/Unload functionality in BSQL has been deprecated. It has been replaced with Mimer SQL LOAD/UNLOAD, see *Chapter 8, Loading and Unloading Data and Definitions*.

Load/Unload is still available from the BSQL menu for backward compatibility.

Readlog from UTIL

The UTIL Readlog functionality has been deprecated. It is now available from BSQL, see *Mimer SQL User's Manual, Chapter 9, READLOG*.

Backup/Restore from UTIL

The Backup/Restore functionality in UTIL has been deprecated. It is now available as SQL statements, see *Mimer SQL Reference Manual, Chapter 12, System Administration Statements*.

Statistics from UTIL

The statistics functionality in UTIL has been deprecated. It is now available as the SQL statement `UPDATE STATISTICS`, see *Mimer SQL Reference Manual, Chapter 12, UPDATE STATISTICS*.

Shadowing Management from UTIL

The UTIL shadowing management has been deprecated. Use the BSQL functionality instead, see *Creating and Managing Shadows* on page *128*.

Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

X

Y