Click here to visit our Web Site

Mimer SQL

User's Manual

Version 8.2

Mimer SQL version 8.2 User's Manual
Second revised edition

December, 2000

Published by Mimer Information Technology AB,
P.O. Box 1713,
SE-751 47 Uppsala, Sweden.
Tel +46(0)18-18 50 00.
Fax +46(0)18-18 51 00.
Internet: http://www.mimer.com

Produced by Mimer Information Technology AB, Uppsala, Sweden.

FOREWORD

Documentation objectives

This manual is intended primarily for users of Mimer SQL who have little or no experience of SQL (Structured Query Language). It describes how to use Mimer SQL for creating and manipulating the database contents without attempting to give an exhaustive description of Mimer SQL.

Refer to the Mimer SQL Reference Manual for a complete syntax description of the statements supported in Mimer SQL.

Also included in this manual is a detailed description of the facilities provided in BSQL.

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 BSQL.

Organization of this manual

This manual is divided into two main sections, dealing respectively with SQL database management facilities and the BSQL interface.

Chapter 1 is a brief introduction to this manual.

Chapters 2-8 describe how to use SQL for database management, and may be used as a guide to SQL for users not familiar with the language:

Chapter 2 presents the general concepts of the Mimer SQL. To a large extent, these concepts are common to other database management systems which support the SQL standards.

Chapter 3 describes how to manage connections (logging on) to a Mimer SQL database.

Chapter 4 describes how to retrieve data from a database using SELECT statements.

Chapter 5 describes how to change the database contents using DELETE, INSERT and UPDATE statements.

Chapter 6 describes transaction handling in the Mimer SQL database system.

Chapter 7 describes how to create database objects (idents, databanks, domains, tables, triggers, modules, functions, procedures etc.).

Chapter 8 describes how to manage privileges in the database.

Chapters 9-11 describe the BSQL facility:

Chapter 9 describes the BSQL facility.

Chapter 10 describes how variables can be handled in BSQL.

Chapter 11 describes error handling in BSQL.

The manual also contains the following appendices:

Appendix A lists the structure and contents of an example database provided with the Mimer SQL distribution and used in the examples in this manual.

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** 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.

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:

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.

For JDBC users:

JDBC information can be found on the internet at the following web addresses: http://java.sun.com/products/jdbc/ and http://www.mimer.com/jdbc/.

For information on specific JDBC methods, please see the online 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.

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

ISO/IEC 9075:1999(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL99.

ISO/IEC 9075:1992(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL92.

ISO/IEC 9075-4:1996(E) Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM). This document contains the standard which specifies the syntax and semantics of a database language for managing and using persistent routines.

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.

Acronyms, terms and trademarks

IEC	International Electrotechnical Commission
ISO	International Standards Organization
SQL	Structured Query Language
PSM	Persistent Stored Modules (i.e. “Stored Procedures”)
X/Open	X/Open is a trademark of the X/Open Company

(All other trademarks are the property of their respective holders.)

CONTENTS

1 INTRODUCTION

Mimer SQL is an advanced database management system developed by Mimer Information Technology AB. 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 for details).

Mimer SQL is available through the following user interfaces:

- BSQL is a line-oriented interface designed for use from command files and scripts. It may also be used in an interactive manner.
- Embedded SQL (ESQL) is used through a host programming language - the programmer writes SQL statements as part of the source code for an application program, which is pre-processed and compiled with the appropriate language-specific facilities. The SQL statements are executed in the context of the application program.
- ODBC is a database independent interface specified by Microsoft. Through ODBC, Mimer SQL can support many of the tools available on the platforms supporting ODBC (e.g. Windows, Unix).
- JDBC™ is a database independent interface for writing database applications in Java™.

This manual provides an introduction to the concepts and usage of Mimer SQL, including its use in the BSQL environment. Embedded SQL is described in the Mimer SQL Programmer's Manual. A full description of the syntax and function of Mimer SQL statements is given in the Mimer SQL Reference Manual.

Note: In the syntax descriptions appearing in this manual, items in square brackets ([]) are optional and items separated by a vertical bar (|) are alternatives. Example: READ [COMMAND | ALL] [INPUT FROM] 'filename'.

2 BASIC CONCEPTS OF MIMER SQL

This chapter provides a general introduction to the basic concepts of Mimer SQL databases and Mimer SQL. It is an important introduction for users who have little or no previous knowledge of the Mimer SQL system or SQL.

2.1 The Mimer SQL relational database

A database is a collection of information organized so that storage, retrieval, and modification of the data is as efficient as possible.

The Mimer SQL database is “relational”, which 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 (see Chapter 3), 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.

2.1.1 The data dictionary

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

- Privileges
- Databanks
- Domains
- Functions
- Idents
- Indexes
- Modules
- Procedures
- Schemas
- Sequences
- Shadows
- Synonyms
- Tables
- Triggers
- Views

The objects stored in a Mimer SQL database can be divided into the following groups:

- **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** belong to a schema. Private object names are local to a schema, so two different schemas may contain an object with the same name. The private objects in a Mimer SQL database are: domains, functions, indexes, modules, procedures, sequences, synonyms, tables, triggers and views.

  Private objects are 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 Section 4.2.3 of the Mimer SQL Reference Manual). 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.

2.1.2 Databanks

A databank is the physical file where a collection of tables is stored. A Mimer SQL database may include any number of databanks. There are two types of 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) and SQLDB (used in transaction handling and for temporary storage of internal work tables).
- **User databanks** contain the user tables. These databanks are defined by the user(s) responsible for setting up the database. (See Section 3.1.2.1 of the Mimer SQL Reference Manual for details concerning pathnames 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 the entire database file base (see Chapter 5 of the Mimer SQL System Management Handbook for more information).

2.1.3 Idents

An ident is an authorization-id used to identify users, programs and groups. There are four types of ident in a Mimer SQL database:

- **User idents** identify individual users who can connect to a Mimer SQL database. A user's access to the database is protected by a password and is 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.
- **OS_USER idents** are idents which reflect a user id defined by the operating system. An OS_USER ident allows the user currently logged in to the operating system to access the Mimer SQL database without providing a username or password. For example: if the current operating system user is ALBERT and there is an OS_USER ident called ALBERT defined in Mimer SQL, ALBERT may start BSQL (for example) and connect directly to Mimer SQL simply by pressing <return> at the Username: prompt. If an OS_USER ident is defined with a password in Mimer SQL, the ident may also connect to Mimer SQL in the same way as a user ident (i.e. by providing the username and password). An OS_USER ident may not have the same name as a user ident in the database.
- **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** 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.

2.1.4 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, OS_USER or PROGRAM ident is created, a schema with the same name can also be automatically 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.

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.

2.1.5 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 on the guests staying at a particular hotel may have columns for the guest's last name, address, check-in and check-out dates:

GUESTS

GUEST_LNAME	ADDRESS	CHECKIN	CHECKOUT
FRANCIS	VIMPELGATAN 7, SKARA	1997-06-19	1997-06-20
LE FEVRE	6 RUE PARISIEN, PARIS,FRA	1997-06-27	1997-07-03
JOHNSSON	DALGATAN 51, SALA	1997-07-14	1997-07-15
PEREZ	CARLOTA 7, MADRID, SPAIN	1997-08-06	-
PERSSON	GROPGATAN 43A, VADSTENA	1997-08-17	-
NYQVIST	KARPV. 33, NYBROVIK	1997-08-18	-
TORP	GRANDV. 77, KRISTIANSTAD	1997-08-19	-

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 been defined for a column, the NULL value (which means the value is unknown) is stored when no data value is supplied.

For example, in the table above, Julio Perez does not have a check-out date listed and the table displays a minus sign in the CHECKOUT column on that row. The minus sign indicates that there is a NULL value stored in the field (the minus sign is how the NULL value is displayed in BSQL, other applications may do it differently).

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 Section 4.3 of the Mimer SQL Reference Manual for a detailed description of data types supported by Mimer SQL.

2.1.6 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 that 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**.

For example, a view may be created on the GUESTS table in the example above to include only GUEST_LNAME and dates for CHECKIN and CHECKOUT:

GUESTS_VIEW

GUEST_LNAME	CHECKIN	CHECKOUT
FRANCIS	1997-06-19	1997-06-20
LE FEVRE	1997-06-27	1997-07-03
JOHNSSON	1997-07-14	1997-07-15
PEREZ	1997-08-06	-
PERSSON	1997-08-17	-
NYQVIST	1997-08-18	-
TORP	1997-08-19	-

Similarly, a view may be created to include only the rows in GUESTS where the CHECKIN column is filled and the CHECKOUT column is NULL (i.e. only guests who are currently staying at the hotel).

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 names and amounts due (as separate items) for guests currently staying at the hotel (bill data is stored in a separate BILL table, linked to GUESTS through the RESERVATION column). Only a portion of the full set of data is shown in this example:

BILL_VIEW

GUEST_LNAME	COST
FIMPLY	100
FIMPLY	70
FIMPLY	-
PEREZ	370
PERSSON	100
...	...
...	...

2.1.7 Unique constraints 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.

Primary key 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).

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.

Indexes are internal structures which cannot be explicitly accessed by the user once created. An index will be used if the internal query optimization process determines it will improve the efficiency of a search.

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.

2.1.8 Routines (functions and procedures)

In Mimer SQL it is possible to define SQL routines that are stored in the data dictionary and which may be invoked when needed. The term “routine” is a collective term for **functions** and **procedures**. The acronym PSM, Persistent Stored Modules, is sometimes used for routines.

For a complete and detailed discussion of functions, procedures and the Stored Procedures functionality supported in Mimer SQL see Chapter 8 of the Mimer SQL Programmer's Manual.

Functions are distinguished from procedures in that they return a single value and have parameters that 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 result set procedures. Result set procedures 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 a CALL statement 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 ident invoking a routine must hold EXECUTE privilege on it.

The creator of a routine must hold the appropriate privileges on any database objects referenced from within the routine. The routine can exist as long as the privileges are held.

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

The PSM constructs available in Mimer SQL allow powerful functionality to be defined and used through the creation and execution of routines. The use of routines also makes it possible to move application logic from the client to the server, thereby reducing network traffic.

2.1.9 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 Chapter 9 of the Mimer SQL Programmer's Manual.

2.1.10 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 privilege 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 (see Sections 7.12 and 8.3.4 for details).

2.1.11 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 ROOM_VIEW is contained in the schema called SAMMY, the full name of the view is SAMMY.ROOM_VIEW.

This view may be referenced from the schema called JIMMY by its fully qualified name as given above.

Alternatively, a synonym may be created for the view in schema JIMMY, e.g. RM_VIEW. Then the name RM_VIEW can simply be used to refer to the view SAMMY.ROOM_VIEW.

Note: The name RM_VIEW is contained in schema JIMMY and can only be used in that context.

2.1.12 Shadows

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

2.1.13 Sequences

A sequence is a private database object that can provide a series of integer values. A sequence can be defined as unique or non-unique.

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

A unique sequence will generate a series of values that change by the increment value from the initial value to a value that does not exceed the maximum value. A unique sequence never generates the same value twice.

A non-unique sequence generates a series of values by starting at the initial value and proceeding in increment steps. If all values in a non-unique sequence has been exhausted, the sequence will start over again with the initial value.

A sequence is created with an **undefined** value initially.

It is possible to generate the next value in the integer series of a sequence by using the “NEXT_VALUE OF *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 OF *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).

When the current value of a **unique** sequence 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 is **non-unique**, “NEXT_VALUE OF *sequence_name*” will always succeed. If the current value of the sequence is equal to the last value in the series it defines, the initial value of the sequence will be returned.

The value of “CURRENT_VALUE OF *sequence_name*” and “NEXT_VALUE OF *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.

<u>Examples</u>:

A **non-unique** sequence with initial value 1, increment 3 and maximum 10 will generate the following series of values: 1, 4, 7, 10, 3, 6, 9, 2, 5, 8, 1, 4, 7… .

A **unique** sequence with initial value 1, increment 3 and maximum 10 will generate the following series of values: 1, 4, 7, 10, 3, 6, 9, 2, 5, 8.

<u>Note:</u> It is possible that not every value in the series defined by the sequence will be generated. If a database server crash etc. occurs during the life of a sequence it is possible that some of the values in the series might be skipped.

2.2 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 four built-in facilities that ensure the data integrity in the database:

- Domains
- Foreign keys (also referred to as referential integrity)
- Check statements 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.

2.2.1 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 and length may be specified explicitly (e.g. CHARACTER(20) or INTEGER(5)) 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, a length specification, optional check conditions and a 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.

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 privilege on the domain (see Section 8.2.2).

2.2.2 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 another 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.

The following example illustrates the column HOTELCODE in table ROOMS as a foreign key referencing the primary key of table HOTEL.

ROOMS

ROOMNO	HOTELCODE	ROOMTYPE	STATUS
LAP110	LAP	SSGLS	FREE
LAP211	LAP	NSDBLB	UNKNOWN
LAP309	LAP	NSSGLS	UNKNOWN
...	...	...	...
...	...	...	...

HOTEL

HOTELCODE	NAME	CITY
LAP	LAPONIA	STOCKHOLM
SKY	SKYLINE	UPPSALA
STG	ST. GEORGE	STOCKHOLM
WINS	WINSTON	GOTHENBURG
WIND	WINSTON	COPENHAGEN
WIN	Winston	London

In this example, the referential constraint means there cannot be a room in a hotel that does not exist, and a hotel cannot be deleted if it has any rooms.

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.

2.2.3 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. For example, the check condition in the definition of the BOOK_GUEST table (see Appendix A) specifies that a guest must be booked to arrive before they depart, and to checkout no earlier than they check in.

Check conditions are discussed in detail in Section 7.5.5.

2.2.4 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 HOTEL_STOCKHOLM is created with the following SQL statement:

```
CREATE  VIEW HOTEL_STOCKHOLM
    AS  SELECT  NAME, CITY
        FROM    HOTEL
        WHERE   CITY = 'STOCKHOLM'
        WITH    CHECK OPTION;
```

This means that the view HOTEL_STOCKHOLM contains NAME and CITY columns from the HOTEL table on the condition that the value in the CITY column is STOCKHOLM. Any attempts to change contents of the CITY column in the view or to insert data in the view where CITY does not contain STOCKHOLM is rejected.

2.3 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 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**, 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, the creator of a function or procedure may execute it, the creator of a program ident may enter it, the creator of a schema may create objects in and drop objects from it, the creator of a databank may create tables in the databank, the creator of a table holds all privileges on the table, the creator of a domain may use that domain and the creator of 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 privileges on objects referenced from within the routine.
- **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.

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. Section 8.3 describes revoking privileges in more detail.

2.4 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, used to maintain objects in a database

CREATE	creates objects
ALTER	modifies objects
DROP	drops objects
COMMENT	documents objects

- access control statements, used to manage privileges

GRANT	grants privileges
REVOKE	revokes privileges

- data manipulation statements, used to examine and change data in the database

SELECT	retrieves data
INSERT	adds new rows to tables
UPDATE	changes data in existing rows
DELETE	deletes data
CALL	executes routines
SET	value assignment

- connection statements, used to connect and disconnect user and program idents to or from the database

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, used to control when database transactions begin and end, and when updates take effect

SET TRANSACTION	set transaction options for subsequent transactions
SET SESSION	set 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, used to manage backup/restore operations and the statistical information used to optimize queries

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

The SQL statements are described in detail in subsequent chapters of this manual and in the Mimer SQL Reference Manual.

In addition, there is a set of commands specific to the BSQL environment, for managing output formatting and so on (see Chapter 9).

Note: In 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.

3 MANAGING DATABASE CONNECTIONS

An application gains access to a Mimer SQL database by establishing a connection to it. A program may have several database connections established simultaneously. Mimer SQL supports the ability to switch between different connections (i.e. access different databases) from within the same application program. Only one database connection is active at any one time.

3.1 Database connections

3.1.1 Connecting to a database

Only idents of type USER and OS_USER can be used to connect to a Mimer SQL database. A connection is established using the CONNECT statement, with the general form (see the Mimer SQL Reference Manual for details):

```
CONNECT TO 'DATABASE' [AS 'CONNECTION_NAME']
      USER 'USER_NAME' USING 'password';
```

This statement establishes a connection between the user and a database.

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 for details. The database can be specified by name or by using the keyword DEFAULT.

Note: If the keyword DEFAULT is used, a user and password cannot be specified - see Chapter 6 of the Mimer SQL Reference Manual. If you wish to connect to the default database **and** specify a user and password, specify an empty string (‘’) for the database.

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.

3.1.2 Changing connections

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. 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.

```
SET CONNECTION 'CONNECTION_NAME';
```

3.1.3 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, 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 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

3.2 Program idents - ENTER and LEAVE

Program idents may be entered from within an SQL session by using the ENTER statement. The current user must have EXECUTE privilege on the program ident in order to perform an ENTER.

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.

Example:

```
ENTER 'PROGRAM_NAME' USING 'secret';

LEAVE RETAIN;
```

The statements ENTER and LEAVE may not be issued within transactions (see Chapter 6).

4 RETRIEVING DATA FROM TABLES

This chapter describes how to retrieve information from the database. In a relational database, information retrieved from one or more *source tables* is returned in the form of a *result table* (sometimes 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 A).

4.1 Retrieval from single tables

4.1.1 Simple retrieval

The simplest retrievals fetch information from one table. The general form of the simple SELECT statement is

```
SELECT column-list FROM table [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.

Find the name and city for all hotels.

```
SELECT  NAME,CITY
FROM    HOTEL;
```

NAME	CITY
LAPONIA	STOCKHOLM
SKYLINE	UPPSALA
ST. GEORGE	STOCKHOLM
Winston	London
WINSTON	COPENHAGEN
WINSTON	GOTHENBURG

Find the name and city for hotels in Stockholm.

```
SELECT  NAME,CITY
FROM    HOTEL
WHERE   CITY='STOCKHOLM';
```

NAME	CITY
LAPONIA	STOCKHOLM
ST. GEORGE	STOCKHOLM

The formulation of selection conditions is described in detail in Section 4.1.4.

The columns in the result table are ordered as they are written in the SELECT statement, irrespective of the ordering in the source table:

```
SELECT   CITY,NAME
FROM     HOTEL;
```

CITY	NAME
STOCKHOLM	LAPONIA
UPPSALA	SKYLINE
STOCKHOLM	ST. GEORGE
London	Winston
COPENHAGEN	WINSTON
GOTHENBURG	WINSTON

A shorthand form for selecting all columns from a table is

```
SELECT * FROM table ...
```

In this case, the columns in the result table are ordered as they are defined in the source table.

Any 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 the qualified form if the schema to which it belongs was not created by the current user, unless it is replaced by a synonym.

Example

```
SELECT   *
FROM     HOTELADM.ROOMTYPES;
```

ROOMTYPE	DESCRIPTION
NSDBLB	NO SMOKING - DOUBLE WITH BATH
NSDBLS	NO SMOKING - DOUBLE WITH SHOWER
NSSGLB	NO SMOKING - SINGLE WITH BATH
NSSGLS	NO SMOKING - SINGLE WITH SHOWER
SDBLB	SMOKING - DOUBLE WITH BATH
SDBLS	SMOKING - DOUBLE WITH SHOWER
SSGLB	SMOKING - SINGLE WITH BATH
SSGLS	SMOKING - SINGLE WITH SHOWER

4.1.2 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).

```
SELECT  NAME AS HOTEL_NAME, CITY AS TOWN
FROM    HOTEL;
```

HOTEL_NAME	TOWN
LAPONIA	STOCKHOLM
SKYLINE	UPPSALA
ST. GEORGE	STOCKHOLM
Winston	London
WINSTON	COPENHAGEN
WINSTON	GOTHENBURG

Labels are particularly useful in queries that retrieve computed values, where the result table column is otherwise unnamed (see Section 4.1.5).

4.1.3 Eliminating duplicate values

The simple SELECT statement retrieves all rows which fulfill the selection conditions. The result table does not have a primary key, and may contain duplicate values.

```
SELECT  RESERVATION, CHARGE_CODE
FROM    BILL;
```

RESERVATION	CHARGE_CODE
1347	100
1347	**120**
1347	210
1347	700
1347	**120**
1348	**700**
1348	**700**
1348	200
1348	230
...	...

Adding the keyword DISTINCT before the column list eliminates all duplicate rows from the result table. The keyword DISTINCT may only be used once in a simple SELECT statement.

```
SELECT  DISTINCT RESERVATION, CHARGE_CODE
FROM    BILL;
```

RESERVATION	CHARGE_CODE
1347	100
1347	**120**
1347	210
1347	700
1348	**700**
1348	200
1348	230
...	...

DISTINCT also eliminates duplicate rows containing NULL values, although technically NULL is not regarded as equal to NULL (see Section 4.3).

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.

4.1.4 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

Comparison operators that may be used in the WHERE clause are:

= 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 > 800 AND PRICE < 1000
```

may not be expressed as

```
WHERE PRICE > 800 AND < 1000
```

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 collating sequence for characters is an extended ASCII character set as defined by ISO 8859-1 (see Appendix B of the Mimer SQL Reference Manual for the exact sequence).

Retrieve the room type, price, and date from which the prices apply for all rooms with hotel code LAP and a cost of under 700.

```
SELECT  ROOMTYPE, PRICE, FROM_DATE, TO_DATE
FROM    ROOM_PRICES
WHERE   HOTELCODE = 'LAP' AND PRICE < 700;
```

ROOMTYPE	PRICE	FROM_DATE	TO_DATE
NSSGLB	660	1997-11-15	1998-03-10
NSSGLS	680	1997-08-08	1997-11-14
NSSGLS	640	1997-11-15	1998-03-10
SSGLB	660	1997-11-15	1998-03-10
SSGLS	680	1997-08-08	1997-11-14
SSGLS	640	1997-11-15	1998-03-10

When stating conditions on temporal data in tables, datetime and interval literals can be specified. There are also the pseudo literals CURRENT_DATE, LOCALTIME and LOCALTIMESTAMP 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.

Retrieve guests who requested a wake up call at 6:00 o'clock today.

```
SELECT   ROOMNO
FROM     WAKE_UP
WHERE    WAKE_DATE = CURRENT_DATE
AND      WAKE_TIME = TIME '06:00:00';
```

ROOMNO
LAP112
SKY111
STG009

Are there any guests scheduled for check in today?

```
SELECT   RESERVED_FNAME, RESERVED_LNAME
FROM     BOOK_GUEST
WHERE    ARRIVE = CURRENT_DATE;
```

RESERVED_FNAME	RESERVED_LNAME
ALEX	OLSSON
BERTIL	GUSTAVSSON
URBAN	FRANSSON

For an example of interval literals, see Section 4.1.13 on datetime arithmetic.

Pattern conditions

LIKE is used to search for character strings that match a specified pattern.

Patterns in the LIKE condition can be written with "wildcard" characters (also called "meta-characters"):

_ (underscore) stands for any single character

% stands for any sequence of zero or more characters

Wildcards only have significance in LIKE predicates.

Find all guests at the Hotel Laponia whose names include "HANSEN" .

```
SELECT   GUEST_LNAME
FROM     BOOK_GUEST
WHERE    GUEST_LNAME LIKE '%HANSEN%' AND HOTELCODE = 'LAP';
```

GUEST_LNAME
JOHANSEN
HANSEN

Find all guests at the Hotel Laponia whose last names do not include "HANSEN".

```
SELECT   GUEST
FROM     BOOK_GUEST
WHERE    GUEST_LNAME NOT LIKE '%HANSEN%' AND HOTELCODE = 'LAP';
```

GUEST_LNAME
DATE
ALVE
KRISTOFERSEN
HOLMER
KULLMER
SMITH
SCHMIDT
ZETTERBERG
HANSSON

Remember that character strings in Mimer SQL statements are always written within apostrophes ('). 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. Thus

```
         'SKYLINE  ' =    'SKYLINE'            is true
         'SKYLINE  ' LIKE 'SKYLINE  '          is true
         'SKYLINE  ' LIKE 'SKYLINE%'           is true
   but   'SKYLINE  ' LIKE 'SKYLINE'            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 for more details.

Some other examples of searching for character strings are:

LIKE '%P%'	matches any string that contains an upper-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

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 hotels are in Stockholm or Copenhagen?

```
SELECT  NAME, CITY
FROM    HOTEL
WHERE   CITY IN ('STOCKHOLM','COPENHAGEN');
```

NAME	CITY
LAPONIA	STOCKHOLM
ST. GEORGE	STOCKHOLM
WINSTON	COPENHAGEN

Which hotels are not in Stockholm or Copenhagen?

```
SELECT  NAME, CITY
FROM    HOTEL
WHERE   CITY NOT IN ('STOCKHOLM','COPENHAGEN');
```

NAME	CITY
SKYLINE	UPPSALA
Winston	London
WINSTON	GOTHENBURG

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 which room types that have prices in the range 700 to 1000 at hotel LAPONIA.

```
SELECT  ROOMTYPE, PRICE
FROM    ROOM_PRICES
WHERE   PRICE BETWEEN 700 AND 1000
AND     HOTELCODE = 'LAP'
```

ROOMTYPE	PRICE
NSDBLB	900
NSDBLB	830
NSDBLS	760
NSDBLS	710
NSDBLS	800
SDBLB	900
SDBLB	830
SDBLS	710
SDBLS	760
SSGLB	800

Find the date, charge code and amount for items billed on dates outside the range 1997-08-30 and 1997-09-01 for the reservation number 1371.

```
SELECT   ON_DATE, CHARGE_CODE, COST
FROM     BILL
WHERE    RESERVATION = 1371
AND      ON_DATE NOT BETWEEN TIMESTAMP '1997-08-30 00:00:00' AND
                             TIMESTAMP '1997-09-01 23:59:59';
```

ON_DATE	CHARGE_CODE	COST
1997-07-06 13:38:19	700	-
1997-07-06 13:38:19	230	200
1997-07-07 13:38:19	100	100
1997-07-08 13:38:19	100	100
1997-07-08 13:38:19	200	-
1997-07-08 13:38:20	230	200
1997-07-09 13:38:20	100	100
1997-07-09 13:38:20	270	95
1997-07-10 13:38:20	100	100
1997-07-10 13:38:20	330	120
1997-07-11 13:38:20	100	100
1997-07-11 13:38:20	200	-
1997-07-12 13:38:20	100	100

BETWEEN may also be used for character comparisons. Strings are compared character by character from left to right.

```
SELECT   NAME
FROM     HOTEL
WHERE    NAME BETWEEN 'SKYLINE' AND 'WINSTON';
```

NAME
SKYLINE
ST. GEORGE
WINSTON
WINSTON

4.1.5 Retrieving computed values

You can retrieve computed values by using arithmetic and string operators in the SELECT clause of the statement. The following computational operators may be used:

+	addition
-	subtraction
*	multiplication
/	division
||	string concatenation

See the Mimer SQL Reference Manual for information regarding the type and precision of the result of an arithmetic expression.

List room prices with a 12% reduction.

```
SELECT  PRICE, PRICE*0.88
FROM    ROOM_PRICES;
```

PRICE	
900	792.00
830	730.40
760	668.80
710	624.80
800	704.00
...	...

The computed column is unnamed by default in the result table. A label may be used to provide a name:

```
SELECT  PRICE, PRICE*0.88 AS SPECIAL_RATE
FROM    ROOM_PRICES;
```

PRICE	SPECIAL_RATE
900	792.00
830	730.40
760	668.80
710	624.80
800	704.00
...	...

A column may also be "computed" as a constant value, which adds an extra column to the result table:

```
SELECT  PRICE, '12% reduction:', PRICE*0.88 AS SPECIAL_RATE
FROM    ROOM_PRICES;
```

PRICE		SPECIAL_RATE
900	12% reduction:	792.00
830	12% reduction:	730.40
760	12% reduction:	668.80
710	12% reduction:	624.80
800	12% reduction:	704.00
...	...	...

You may also retrieve a value computed using the values in two or more columns, providing that the data types are compatible.

Retrieve hotel names prefixed with the word "HOTEL " and cities.

```
SELECT  'HOTEL ' || NAME, CITY
FROM    HOTEL;
```

	CITY
HOTEL LAPONIA	STOCKHOLM
HOTEL SKYLINE	UPPSALA
HOTEL ST. GEORGE	STOCKHOLM
HOTEL Winston	London
HOTEL WINSTON	COPENHAGEN
HOTEL WINSTON	GOTHENBURG

For string concatenation, column values are padded with trailing blanks to the length of the column definition. For example:

```
SELECT   NAME || 'HOTEL', CITY
FROM     HOTEL;
```

		CITY
LAPONIA	HOTEL	STOCKHOLM
SKYLINE	HOTEL	UPPSALA
ST. GEORGE	HOTEL	STOCKHOLM
Winston	HOTEL	London
WINSTON	HOTEL	COPENHAGEN
WINSTON	HOTEL	GOTHENBURG

When retrieving computed values, parentheses can be used to force the operation priority. Without parentheses, the normal precedence rules for arithmetic apply, i.e. multiplication and division are performed before addition and subtraction, and operators with the same precedence are evaluated from left to right.

4.1.6 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 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 Section 4.1.7).

AVG average of values (numerical columns only)

COUNT number of rows

MAX largest value

MIN smallest value

SUM sum of values (numerical columns only)

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.

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 Section 4.1.7). Thus

```
SELECT   COUNT(HOTELCODE), NAME, CITY
FROM     HOTEL;
```

is illegal.

The set functions are illustrated with results from the table

SAMPLE
1.0
2.0
2.0
2.0
3.0
3.0
4.0
5.0
-
-

(A hyphen "-" indicates NULL).

```
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).
```

Some further examples of set functions applied to the example database are given below.

How many rows are there in the BOOK_GUEST table?

```
SELECT  COUNT(*)
FROM    BOOK_GUEST;
```

How many guests have checked out (i.e. CHECKOUT is not NULL)?

```
SELECT  COUNT(ALL CHECKOUT)
FROM    BOOK_GUEST;
```

What is the total bill for reservation number 1359.

```
SELECT  SUM(COST)
FROM    BILL
WHERE   RESERVATION = 1359;
```

Find the average price of NO SMOKING single rooms in the hotel chain.

```
SELECT  AVG(PRICE)
FROM    ROOM_PRICES
WHERE   ROOMTYPE IN ('NSSGLB','NSSGLS');
```

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)) ...
```

4.1.7 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 most expensive NO SMOKING single room in each hotel.

```
SELECT  HOTELCODE, MAX(PRICE) AS EXPENSIVE
FROM    ROOM_PRICES
WHERE   ROOMTYPE = 'NSSGLB'
OR      ROOMTYPE = 'NSSGLS'
GROUP BY HOTELCODE;
```

HOTELCODE	EXPENSIVE
LAP	800
SKY	870
STG	680
WIND	1410
WINS	1370

Using a GROUP BY clause places some restrictions on the SELECT statement:

- Only constants, columns used in the GROUP BY clause, and columns used in set functions may be included in the SELECT list
- A column used in the GROUP BY clause may not be used in a set function.

How many hotels are there in each city?

```
SELECT  CITY, COUNT(HOTELCODE)
FROM    HOTEL
GROUP BY CITY;
```

CITY	
COPENHAGEN	1
GOTHENBURG	1
London	1
STOCKHOLM	2
UPPSALA	1

In a statement with column references in the SELECT list, all columns not used in set functions must be used as grouping columns.

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
-
-

```
SELECT  SAMPLE, COUNT(*) AS NUMBER
...
GROUP BY SAMPLE;
```

SAMPLE	NUMBER
1.0	1
2.0	3
3.0	2
4.0	1
5.0	1
-	2

4.1.8 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 highest price for a SMOKING single room in each hotel, but restrict the selection to prices over 1000.

```
SELECT  HOTELCODE, MAX(PRICE)
FROM    ROOM_PRICES
WHERE   ROOMTYPE = 'SSGLB'
OR      ROOMTYPE = 'SSGLS'
GROUP BY HOTELCODE
HAVING  MAX(PRICE) > 1000;
```

HOTELCODE	
WIND	1410
WINS	1370

4.1.9 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. (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. A change in the database contents may otherwise change the order, particularly for a complex query where the order of execution is determined by the SQL optimizer).

Retrieve the hotel code, room type, from date and price for SMOKING single rooms with showers with a cost of under 800 and order by the price in descending order.

```
SELECT   *
FROM     ROOM_PRICES
WHERE    PRICE < 800
AND      ROOMTYPE = 'SSGLS'
ORDER BY PRICE DESC;
```

HOTELCODE	ROOMTYPE	FROM_DATE	TO_DATE	PRICE
SKY	SSGLS	1997-08-08	1997-11-14	750
STG	SSGLS	1997-08-08	1997-11-14	680
LAP	SSGLS	1997-08-08	1997-11-14	680
STG	SSGLS	1997-11-15	1998-03-10	640
LAP	SSGLS	1997-11-15	1998-03-10	640

More than one column may be specified in the ORDER BY clause:

```
SELECT   *
FROM     ROOM_PRICES
WHERE    PRICE < 800
AND      ROOMTYPE = 'NSSGLS'
ORDER BY HOTELCODE, PRICE;
```

HOTELCODE	ROOMTYPE	FROM_DATE	TO_DATE	PRICE
LAP	NSSGLS	1997-11-15	1998-03-10	640
LAP	NSSGLS	1997-08-08	1997-11-14	680
SKY	NSSGLS	1997-08-08	1997-11-14	750
STG	NSSGLS	1997-11-15	1998-03-10	640
STG	NSSGLS	1997-08-08	1997-11-14	680

To order a result table by a set function or computed value, the column in the result table is given a label and the label is used in the ORDER BY clause:

```
SELECT   ROOMTYPE, AVG(PRICE) AS AVERAGE_PRICE
FROM     ROOM_PRICES
GROUP BY ROOMTYPE
ORDER BY AVERAGE_PRICE;
```

ROOMTYPE	AVERAGE_PRICE
NSSGLS	793
SSGLS	793
NSDBLS	910
NSSGLB	910
SDBLS	910
SSGLB	910
NSDBLB	1128
SDBLB	1128

The following formulation is incorrect, since there is no PRICE column in the result table by which to perform the ordering:

```
SELECT   ROOMTYPE, AVG(PRICE)
FROM     ROOM_PRICES
GROUP BY ROOMTYPE
ORDER BY PRICE;
```

4.1.10 Using scalar functions

These functions operate on expressions or on a single value received from a SELECT statement.

Some of the standard scalar functions available are (the complete list of scalar functions can be found in the Mimer SQL Reference Manual):

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

See the Mimer SQL Reference Manual for the syntax rules and for information regarding the data type of the result of the scalar functions.

Here follows some examples in order to illustrate how the scalar functions may be used:

List all hotels with name Winston, spelled with either upper or lower case letters.

```
SELECT   NAME,CITY
FROM     HOTEL
WHERE    UPPER(NAME) = 'WINSTON';
```

NAME	CITY
Winston	London
WINSTON	COPENHAGEN
WINSTON	GOTHENBURG

List all double rooms at hotel SKY.

```
SELECT  ROOMNO,ROOMTYPE
FROM    ROOMS
WHERE   SUBSTRING(ROOMTYPE FROM 3 FOR 3) = 'DBL'
AND     HOTELCODE = 'SKY';
```

ROOMNO	ROOMTYPE
SKY121	NSDBLS
SKY124	NSDBLB
SKY125	NSDBLB
SKY212	NSDBLB

Get name and address (without trailing blanks) of guest with reservation number 1348.

```
SELECT  TRIM(TRAILING FROM GUEST_LNAME) ||
        ', ' ||
        TRIM(TRAILING FROM ADDRESS)
FROM    BOOK_GUEST
WHERE   RESERVATION = 1348;
```

JOHANSEN, MIMERGATAN 4, UPPSALA

Remove leading and trailing spaces and get length (no. of characters) of description and the description (in lower case) for all charges.

```
SELECT  CHAR_LENGTH(TRIM(DESCRIPTION)), LOWER(TRIM(DESCRIPTION))
FROM    CHARGES;
```

7	lodging
9	telephone
8	car park
10	restaurant
7	minibar
3	bar
12	room service
7	laundry
4	room
9	extra bed
13	miscellaneous

List all the guest names that sounds like "Johnson".

```
SELECT  GUEST_LNAME
FROM    BOOK_GUEST
WHERE   SOUNDEX(GUEST_LNAME) = SOUNDEX('JOHNSON');
```

JANSSON
JONSON
JOHNZON

4.1.11 Using 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 Section 5.6 of the Mimer SQL Reference Manual. The following select statements presents two examples of how CASE expressions can be used:

Translate the currency code in the exchange_rate table to descriptive names.

```
SELECT CASE CURRENCY
         WHEN 'DEM' THEN 'German Marks'
         WHEN 'DKK' THEN 'Danish Crowns'
         WHEN 'FRF' THEN 'French Francs'
         WHEN 'GBP' THEN 'British Pounds'
         WHEN 'ITL' THEN 'Italian Lira'
         ELSE CURRENCY
       END  AS CURRENCY, RATE
FROM   EXCHANGE_RATE;
```

CURRENCY	RATE
German Marks	0.223
Danish Crowns	0.849
FIM	0.656
French Francs	0.742
British Pounds	0.081
Italian Lira	206.820
JPY	16.380
NOK	0.881
SEK	1.000
USD	0.133

This form of a case expression is known as a simple case expression, in which an operand (CURRENCY in this case) 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.

The other form of the case expression can be seen in the following example:

Divide room prices into different categories.

```
SELECT CASE
         WHEN PRICE >= 900 then 'Expensive'
         WHEN PRICE <= 700 then 'Budget'
         ELSE 'Moderate'
       END  AS CATEGORY, ROOMTYPE, PRICE
FROM   ROOM_PRICES;
```

CATEGORY	ROOMTYPE	PRICE
Expensive	NSDBLB	900
...		
Budget	NSSGLB	660
...		
Moderate	SDBLB	830
...		

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.

4.1.12 Using 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 Mimer SQL Reference Manual for conversion rules.

List the billed charges for reservation number 1347. Convert the charged amounts to US-dollars to decimal with scale 4. Convert the date of charges (in format YYYY-MM-DD) to character in format DD/MM/YY.

```
SELECT CAST(CHARGE_CODE AS SMALLINT) AS CODE,
       CAST(COST/7.835 AS DECIMAL(10,4)) AS USD,
       SUBSTRING(CAST(ON_DATE AS CHAR(26)) FROM 9 FOR 2)||'/'||
       SUBSTRING(CAST(ON_DATE AS CHAR(26)) FROM 6 FOR 2)||'/'||
       SUBSTRING(CAST(ON_DATE AS CHAR(26)) FROM 3 FOR 2) AS DATE
FROM   BILL
WHERE  RESERVATION = 1347
ORDER BY CODE;
```

CODE	USD	DATE
100	12.7632	21/08/97
120	5.1052	21/08/97
120	5.1052	21/08/97
210	-	21/08/97
700	-	21/08/97

4.1.13 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.

How many days have the guests at hotel LAPONIA stayed?

```
SELECT  GUEST_LNAME,
        (COALESCE(CHECKOUT,CURRENT_DATE)-CHECKIN) DAY(2) AS DAYS
FROM    BOOK_GUEST
WHERE   HOTELCODE = 'LAP'
AND     CHECKIN IS NOT NULL;
```

GUEST_LNAME	DAYS
DATE	1
JOHANSEN	2
HANSEN	1
ALVE	2
KRISTOFFERSEN	1
HOLMER	4
...	...
ZETTERBERG	3
HANSSON	6

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 example above. In that example the precision is actually superfluous as the default precision for day is 2.

The above example uses the COALESCE short form of the CASE expression, a complete description of this can be found in Section 5.6 of the Mimer SQL Reference Manual.

Which hotel rooms have requested a wake up call within the next hour and a half (assuming the time is 08:35:00)?

```
SELECT   ROOMNO
FROM     WAKE_UP
WHERE    WAKE_DATE = CURRENT_DATE
AND      WAKE_TIME BETWEEN LOCALTIME AND
           LOCALTIME + INTERVAL '01:30' HOUR TO MINUTE;
```

ROOMNO
SKY101
SKY201

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.

It is possible to extract part of a datetime value with the EXTRACT function. The function returns a numeric value.

Which month did FREDRIK SELLIN stay at any of the hotels?

```
SELECT CASE EXTRACT (MONTH FROM ARRIVE)
         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  AS MONTH
FROM   BOOK_GUEST
WHERE  GUEST_FNAME = 'FREDRIK' AND GUEST_LNAME = 'SELLIN';
```

MONTH
JULY

Another useful function is DAYOFWEEK which returns the day number within a week. MONDAY has the value 1 and SUNDAY has the value 7.

Which day did FREDRIK SELLIN arrive at any of the hotels?

```
SELECT CASE DAYOFWEEK(ARRIVE)
         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  AS DAY
FROM   BOOK_GUEST
WHERE  GUEST_FNAME = 'FREDRIK' AND GUEST_LNAME = 'SELLIN';
```

DAY
SUNDAY

4.2 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.

4.2.1 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.

List the billed charges for reservation number 1349.

```
SELECT  DESCRIPTION, COST
FROM    CHARGES, BILL
WHERE   RESERVATION = 1349
AND     BILL.CHARGE_CODE = CHARGES.CHARGE_CODE;
```

The join condition here is BILL.CHARGE_CODE = CHARGES.CHARGE_CODE, which relates the charge code in table BILL (where amounts are listed) to the charge code in table CHARGES (where the text description of the charge code is listed). The result is:

DESCRIPTION	COST
ROOM	-
CAR PARK	70
MISCELLANEOUS	30

Conceptually, the join first establishes a table containing all combinations of the rows in CHARGES with the rows in BILL, then selects those rows in which the two CHARGE_CODE values are equal (see Section 4.4 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.

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:

```
SELECT  DESCRIPTION, COST
FROM    CHARGES, BILL
WHERE   RESERVATION = 1349;
```

DESCRIPTION	COST
LODGING	-
TELEPHONE	-
CAR PARK	-
RESTAURANT	-
MINIBAR	-
BAR	-
ROOM SERVICE	-
LAUNDRY	-
ROOM	-
EXTRA BED	-
MISCELLANEOUS	-
LODGING	70
TELEPHONE	70
CAR PARK	70
RESTAURANT	70
MINIBAR	70
BAR	70
ROOM SERVICE	70
LAUNDRY	70
ROOM	70
EXTRA BED	70
MISCELLANEOUS	70
LODGING	30
TELEPHONE	30
CAR PARK	30
RESTAURANT	30
MINIBAR	30
BAR	30
ROOM SERVICE	30
LAUNDRY	30
ROOM	30
EXTRA BED	30
MISCELLANEOUS	30

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 BSQL is used), where queries are so easily written and submitted.

4.2.2 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 Section 4.2.4).

An example of a simple join is the query described in Section 4.2.1:

```
SELECT  DESCRIPTION, COST
FROM    CHARGES, BILL
WHERE   BILL.CHARGE_CODE = CHARGES.CHARGE_CODE
AND     RESERVATION = 1349;
```

DESCRIPTION	COST
ROOM	-
CAR PARK	70
MISCELLANEOUS	30

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 usually duplicated:

```
SELECT  *
FROM    CHARGES, BILL
...;
```

(From CHARGES) *(From BILL)*

CHARGE_CODE	DESCRIPTION	CHARGE_PRICE	RESERVATION	ON_DATE	**CHARGE_CODE**	COST
...	...	...	...	...	...	...

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.

There is an alternative formulation of the query above:

```
SELECT  DESCRIPTION, COST
FROM    CHARGES JOIN BILL
ON      CHARGES.CHARGE_CODE = BILL.CHARGE_CODE
AND     RESERVATION = 1349;
```

All predicates that can be used in a where clause, except sub-selects, can be used in an on-clause. The join clause can be used as a statement on it's own:

```
CHARGES JOIN BILL ON CHARGES.CHARGE_CODE = BILL.CHARGE_CODE;
```

or

```
CHARGES NATURAL JOIN BILL;
```

A natural join, joins the table on the condition of equality between any columns with the same name, in the two tables. In the first example, all columns from the two tables are present in the result. In the second example the join columns will only occur once. Thus, in the first case, the CHARGE_CODE column appears twice in the result, while there is only one occurrence of this column in the second result.

It is possible to nest join-clauses:

Select the status of all rooms at hotel LAPONIA.

```
SELECT  ROOMNO, STATUS
FROM    ROOMSTATUS NATURAL JOIN ROOMS
JOIN    HOTEL
ON      HOTEL.HOTELCODE = ROOMS.HOTELCODE
AND     HOTEL.NAME = 'LAPONIA';
```

ROONO	STATUS
LAP110	FREE
LAP111	UNKNOWN
LAP112	FREE
LAP120	UNKNOWN
LAP121	UNKNOWN
LAP122	UNKNOWN
LAP200	UNKNOWN
LAP201	UNKNOWN
LAP205	FREE
LAP206	UNKNOWN
LAP210	UNKNOWN
LAP211	UNKNOWN
LAP212	UNKNOWN
LAP301	FREE
LAP302	FREE
LAP303	UNKNOWN
LAP304	UNKNOWN
LAP305	UNKNOWN
LAP306	UNKNOWN
LAP307	FREE
LAP308	KEY OUT
LAP309	UNKNOWN

The natural join between ROOMSTATUS and ROOMS is slightly contrived in this example and is present to demonstrate that joins can be nested. If the STATUS column in the ROOMS table was not a foreign key referencing the ROOMSTATUS table, the function of the join could be to validate values in the ROOMS.STATUS column.

A join query can join any number of tables, using complex search conditions to select the relevant information from each table:

Select the total bill for guest Sten Johansen and list it in both Swedish and Danish crowns (SEK and DKK respectively).

```
SELECT   GUEST_LNAME, SUM(COST)/RATE AS TOTAL_BILL, CURRENCY
FROM     BOOK_GUEST, BILL, EXCHANGE_RATE
WHERE    GUEST_LNAME = 'JOHANSEN'
AND      (CURRENCY = 'DKK'
OR        CURRENCY = 'SEK')
AND      BOOK_GUEST.RESERVATION = BILL.RESERVATION
GROUP BY GUEST_LNAME, CURRENCY, RATE;
```

GUEST_LNAME	TOTAL_BILL	CURRENCY
JOHANSEN	235.571	DKK
JOHANSEN	200.000	SEK

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.

4.2.3 Outer joins

The joins in the previous chapter 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.

```
SELECT   DESCRIPTION, COST
FROM     CHARGES LEFT OUTER JOIN BILL
ON       CHARGES.CHARGE_CODE = BILL.CHARGE_CODE
AND      RESERVATION = 1349;
```

DESCRIPTION	COST
LODGING	-
TELEPHONE	-
CAR PARK	70
RESTAURANT	-
MINIBAR	-
BAR	-
ROOM SERVICE	-
LAUNDRY	-
ROOM	-
EXTRA BED	-
MISCELLANEOUS	30

In this example, all rows from the table to the left in the join clause, i.e. CHARGES, are present in the result. Non-matching rows from the BILL table are filled with null values in the result.

Observe the difference in result for the next statement and the previous one.

```
SELECT   DESCRIPTION, COST
FROM     CHARGES LEFT OUTER JOIN BILL
ON       CHARGES.CHARGE_CODE = BILL.CHARGE_CODE
WHERE    RESERVATION = 1349;
```

DESCRIPTION	COST
CAR PARK	70
ROOM	-
MISCELLANEOUS	30

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.

A right outer join will take all records from the table to the right in the join-clause.

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, and not the right table in the first join-clause.

4.2.4 Nested selects

A form of SELECT, called a *subselect*, 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 names of hotels which have rooms with a price under 750.

```
SELECT   NAME
FROM     HOTEL
WHERE    HOTELCODE IN (SELECT   HOTELCODE
                       FROM     ROOM_PRICES
                       WHERE    PRICE < 750 );
```

NAME
LAPONIA
ST. GEORGE

To see how this works, evaluate the subselect first:

```
SELECT   HOTELCODE
FROM     ROOM_PRICES
WHERE    PRICE < 750;
```

HOTELCODE
LAP
LAP
LAP
LAP
LAP
LAP
LAP
LAP
STG
STG
STG
STG
STG
STG
STG
STG
STG
STG

Then use the result of the subselect in the search condition of the outer select:

```
SELECT   NAME
FROM     HOTEL
WHERE    HOTELCODE IN ('LAP','STG');
```

NAME
LAPONIA
ST. GEORGE

A subselect can be used in a search condition wherever the result of the subselect can provide the correct form of the data for the search condition.

Thus a subselect used with "=" must give a single value as a result, a subselect used with IN, ALL or ANY must give a set of single values (see Section 4.2.8) and a subselect used with EXISTS may give any result (see Section 4.2.7).

```
WHERE column = (subselect)
WHERE column IN (subselect)
WHERE column = ALL (subselect)
WHERE column = ANY (subselect)
WHERE EXISTS (subselect)
```

Subselects cannot include ORDER BY clauses. The UNION operator can be used to combine two or more subselects in more complex statements (see Section 4.2.9).

Many nested queries can equally well be written as simple joins. For example:

Select the names of hotels which have rooms with a price under 750.

```
SELECT  NAME
FROM    HOTEL
WHERE   HOTELCODE IN (SELECT  HOTELCODE
                      FROM    ROOM_PRICES
                      WHERE   PRICE < 750 );
```

or alternatively

```
SELECT  DISTINCT NAME
FROM    HOTEL, ROOM_PRICES
WHERE   HOTEL.HOTELCODE = ROOM_PRICES.HOTELCODE
AND     ROOM_PRICES.PRICE < 750;
```

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.

Queries may contain any number of subselects, for example:

List hotels which have rooms that are more expensive than any of the rooms at the Hotel Laponia.

```
SELECT  NAME
FROM    HOTEL
WHERE   HOTELCODE IN
          (SELECT  HOTELCODE
           FROM    ROOM_PRICES
           WHERE   PRICE >
                     (SELECT  MAX(PRICE)
                      FROM    ROOM_PRICES
                      WHERE   HOTELCODE =
                                (SELECT  HOTELCODE
                                 FROM    HOTEL
                                 WHERE   NAME = 'LAPONIA')));
```

(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 NAME
FROM    HOTEL, ROOM_PRICES
WHERE   HOTEL.HOTELCODE = ROOM_PRICES.HOTELCODE
AND     PRICE > (SELECT  MAX(PRICE)
                 FROM    ROOM_PRICES, HOTEL
                 WHERE   ROOM_PRICES.HOTELCODE = HOTEL.HOTELCODE
                 AND     NAME = 'LAPONIA' );
```

4.2.5 Ordering nested queries

The ORDER BY clause may only be used in outer SELECT statements and not in subselects.

The following example is correct:

```
SELECT   NAME, ROOMTYPE, FROM_DATE, PRICE
FROM     HOTEL, ROOM_PRICES
WHERE    ROOMTYPE IN ('NSSGLS','NSSGLB')
ORDER BY NAME;
```

The following example is incorrect:

```
SELECT   NAME, ROOMTYPE, FROM_DATE, PRICE
FROM     HOTEL, ROOM_PRICES
WHERE    HOTEL.HOTELCODE IN (SELECT   HOTELCODE
                             FROM     ROOM_PRICES
                             WHERE    ROOMTYPE IN ('NSSGLS','NSSGLB')
                             ORDER BY HOTELCODE);
```

4.2.6 Correlation names

A correlation name is a temporary name given to a table to represent a logical copy of the table within a query. Correlation names can be up to a maximum of 128 characters long.

There are three uses for correlation names:

- simplifying complex queries
- joining a table to itself
- outer references in subselects

4.2.6.1 Simplifying complex queries

Using short correlation names into complicated queries can make the query easier to write and understand, particularly when qualified table names are used:

```
SELECT HOTELADM.BOOK_GUEST.GUEST_LNAME,
       HOTELADM.HOTEL.NAME, SUM(COST)
FROM   HOTELADM.BOOK_GUEST, HOTELADM.HOTEL, HOTELADM.BILL
WHERE  HOTELADM.BILL.RESERVATION = HOTELADM.BOOK_GUEST.RESERVATION
AND    HOTELADM.HOTEL.HOTELCODE = 'WINS'
GROUP BY HOTELADM.BOOK_GUEST.GUEST_LNAME, HOTELADM.HOTEL.NAME;
```

may be rewritten

```
SELECT   G.GUEST_LNAME, H.NAME, SUM(COST)
FROM     HOTELADM.BOOK_GUEST AS G,
         HOTELADM.HOTEL      AS H,
         HOTELADM.BILL       AS B
WHERE    B.RESERVATION = G.RESERVATION
AND      H.HOTELCODE = 'WINS'
GROUP BY G.GUEST_LNAME, H.NAME;
```

The keyword AS in the FROM clause may be omitted, but is recommended for clarity. Do not confuse AS in the FROM clause (defining a correlation name) with AS in the select list (see Section 4.1.2, defining a label).

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   HOTELADM.BOOK_GUEST AS G,
...
WHERE  H.RESERVATION = HOTELADM.BOOK_GUEST.RESERVATION
```

4.2.6.2 Joining a table with itself

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 pairs of hotels located in the same city.

```
SELECT  HOTEL.NAME, HOTEL.CITY
FROM    HOTEL, HOTEL AS COPY
WHERE   HOTEL.CITY = COPY.CITY
AND     HOTEL.NAME <> COPY.NAME;
```

NAME	CITY
LAPONIA	STOCKHOLM
ST. GEORGE	STOCKHOLM

Here, the table HOTEL is joined to a logical copy of itself called COPY. The first search condition finds pairs of hotels in the same city, and the second eliminates “pairs” with the same name. (Without the second condition in the search condition, all hotel names would be selected!)

Without correlation names, this kind of query cannot be formulated. The following query would select all the hotel names from the table:

```
SELECT  HOTEL.NAME, HOTEL.CITY
FROM    HOTEL
WHERE   HOTEL.CITY = HOTEL.CITY;
```

4.2.6.3 Outer references in subselects

In some constructions using subselects, a subselect 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  NAME
FROM    HOTEL
WHERE   EXISTS (SELECT  *
                FROM    BOOK_GUEST
                WHERE   HOTELCODE = HOTEL.HOTELCODE);
```

This kind of query processes the subselect 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 HOTEL, select the NAME column if there are rows in BOOK_GUEST containing the current HOTELCODE value”.

If the qualifying name in an outer reference is not unambiguous in the context of the subselect, 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  NAME
FROM    HOTEL AS H
WHERE   EXISTS (SELECT  *
                FROM    BOOK_GUEST
                WHERE   HOTELCODE = H.HOTELCODE);
```

4.2.7 Retrieving with 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 subselect.

The subselect 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 subselect is empty or not - a specified column is seldom relevant in the subquery.

EXISTS (subselect)	is true if the result set of the subselect is not empty
NOT EXISTS (subselect)	is true if the result set of the subselect is empty

SELECT statements with EXISTS almost always include an outer reference linking the subselect to the outer select.

Find the names of hotels for which guests exist in the BOOK_GUEST table.

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

Without the outer reference, the select becomes a conditional “all-or-nothing” statement: perform the outer select if the subselect result is not empty, otherwise select nothing.

List all reservation numbers if anybody has checked out without paying.

```
SELECT  DISTINCT RESERVATION
FROM    BILL
WHERE   EXISTS (SELECT  *
                FROM    BOOK_GUEST
                WHERE   CHECKOUT IS NOT NULL
                AND     PAYMENT  IS NULL);
```

The next example illustrates NOT EXISTS:

Which hotels do not have double rooms with showers?

```
SELECT   NAME, HOTELCODE
FROM     HOTEL AS H
WHERE    NOT EXISTS (SELECT  *
                     FROM    ROOMS
                     WHERE   HOTELCODE = H.HOTELCODE
                     AND     ROOMTYPE IN ('NSDBLS','SDBLS');
```

NAME	HOTELCODE
WINSTON	WINS
Winston	WIN

Negated EXISTS clauses must be handled with care. There are two semantic “opposites” to EXISTS, with very different meanings:

```
WHERE EXISTS (SELECT  *
              FROM    GUESTS
              WHERE   GUEST = 'CODD')
```

is true if at least one guest is called CODD.

```
WHERE NOT EXISTS (SELECT  *
                  FROM    GUESTS
                  WHERE   GUEST = 'CODD')
```

is true if no guest is called CODD.

But

```
WHERE EXISTS (SELECT  *
              FROM    GUESTS
              WHERE   GUEST <> 'CODD')
```

is true if at least one guest is not called CODD.

```
WHERE NOT EXISTS (SELECT  *
                  FROM    GUESTS
                  WHERE   GUEST <> 'CODD')
```

is true if no guest is not called CODD, that is, if every guest is called CODD.

The double negative in the previous example is an SQL implementation of the universal quantifier FORALL (see “A Guide to DB2” by C. J. Date for more information on EXISTS and FORALL).

4.2.8 Retrieval with ALL, ANY, SOME

Subselects that return a set of values may be used in the quantified predicates ALL, ANY or SOME. Thus

```
WHERE PRICE < ALL (subselect)
```

selects rows where the price is less than **every** value returned by the subselect

```
WHERE PRICE < ANY (subselect)
```

selects rows where the price is less than **at least one** of the values returned by the subselect

Select room types and hotel codes for rooms with a price that differs from that of each room at Hotel Skyline.

```
SELECT  DISTINCT ROOMTYPE, HOTELCODE
FROM    ROOM_PRICES
WHERE   PRICE <> ALL (SELECT  PRICE
                      FROM    ROOM_PRICES
                      WHERE   HOTELCODE = 'SKY');
```

If the result of the subselect 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 sub-select, is to use EXISTS or NOT EXISTS to see if values are returned by a sub-select which only selects for specific values.

For example:

Select the room type, price and hotel code for rooms which have the same price as a room at the hotel Skyline.

```
SELECT  ROOMTYPE, PRICE, HOTELCODE
FROM    ROOM_PRICES
WHERE   PRICE = ANY (SELECT  PRICE
                     FROM    ROOM_PRICES
                     WHERE   HOTELCODE = 'SKY');
```

is equivalent to

```
SELECT  ROOMTYPE, PRICE, HOTELCODE
FROM    ROOM_PRICES RP
WHERE   EXISTS (SELECT  *
                FROM    ROOM_PRICES
                WHERE   HOTELCODE = 'SKY'
                AND     RP.PRICE = PRICE);
```

4.2.9 Union queries

The UNION operator combines the results of two or more subselect clauses. UNION first merges the result tables specified by the separate subselects and then eliminates duplicate rows from the merged set.

Select the codes for hotels which are in Stockholm or have single rooms with showers.

```
SELECT  HOTELCODE
FROM    HOTEL
WHERE   CITY = 'STOCKHOLM'

UNION

SELECT  DISTINCT HOTELCODE
FROM    ROOMS
WHERE   ROOMTYPE IN ('NSSGLS','SSGLS');
```

The result is obtained by merging the results of the two subselects and eliminating duplicates:

```
SELECT   HOTELCODE
FROM     HOTEL
WHERE    CITY = 'STOCKHOLM';
```

HOTELCODE
LAP
STG

```
SELECT   DISTINCT HOTELCODE
FROM     ROOMS
WHERE    ROOMTYPE IN ('NSSGLS','SSGLS');
```

HOTELCODE
LAP
SKY
STG
WIND

giving the result table

HOTELCODE
LAP
SKY
STG
WIND

To retain duplicates in the result table, use UNION ALL in place of UNION (see the Mimer SQL Reference Manual for details).

Columns which are merged by UNION must have compatible data types (numerical with numerical, character with character). Subselects addressing more than one result column are merged column by column in the order of selection. The number of columns addressed in each subselect must be the same.

The column names in the result of a UNION are taken from the names in the first subselect. Use labels in the first subselect to assign different column names to the result table:

Merge the codes and names of hotels in Stockholm with the hotel codes and room type for rooms which are more expensive than any room at the St. George hotel.

```
SELECT   HOTELCODE AS CODE, NAME AS NAME_OR_TYPE
FROM     HOTEL
WHERE    CITY = 'STOCKHOLM'

UNION

SELECT   HOTELCODE, ROOMTYPE
FROM     ROOM_PRICES
WHERE    PRICE > (SELECT   MAX(PRICE)
                  FROM     ROOM_PRICES
                  WHERE    HOTELCODE = 'STG');
```

CODE	NAME_OR_TYPE
LAP	LAPONIA
STG	ST. GEORGE
WIND	NSDBLB
WIND	NSDBLS
WIND	NSSGLB
WIND	NSSGLS
WIND	SDBLB
WIND	SDBLS
WIND	SSGLB
WIND	SSGLS
WINS	NSDBLB
WINS	NSSGLB
WINS	SDBLB
WINS	SSGLB

Subselects merged by UNION may not include an ORDER BY clause. However, the result of the UNION query may be ordered with an ORDER BY clause placed after the last query in the UNION.

UNION may not be used within a nested subselect. However, the results of nested queries may be joined by UNION.

Unions can also be used to combine information from the same table:

Find the highest and lowest prices for rooms at the Hotel Skyline.

```
SELECT  'HIGHEST' AS PRICE, MAX(PRICE) AS AMOUNT
FROM    ROOM_PRICES
WHERE   HOTELCODE = 'SKY'

UNION

SELECT  'LOWEST', MIN(PRICE)
FROM    ROOM_PRICES
WHERE   HOTELCODE = 'SKY'
ORDER BY AMOUNT;
```

PRICE	AMOUNT
LOWEST	750
HIGHEST	1080

Unions can also be used to perform *outer joins*, joining information in a table or tables with information not listed in those tables (i.e. information that is null). For example:

List the room types available for each hotel code. Include a row for hotel codes which do not have a given room type with a shower.

```
SELECT   DISTINCT H.HOTELCODE, ROOMTYPE
FROM     ROOMS R, HOTEL H
WHERE    R.HOTELCODE = H.HOTELCODE

UNION

SELECT   DISTINCT H.HOTELCODE, 'NO '|| ROOMTYPE AS ROOMTYPE
FROM     HOTEL H, ROOMS
WHERE    H.HOTELCODE = ROOMS.HOTELCODE
AND      NOT EXISTS (SELECT   *
                     FROM     ROOMS R
                     WHERE    R.HOTELCODE = H.HOTELCODE
                     AND      ROOMTYPE LIKE '%S')
ORDER BY HOTELCODE;
```

HOTELCODE	ROOMTYPE
LAP	NSDBLB
LAP	NSDBLS
LAP	NSSGLB
LAP	NSSGLS
LAP	SDBLS
LAP	SSGLB
LAP	SSGLS
SKY	NSDBLB
SKY	NSDBLS
SKY	NSSGLB
SKY	NSSGLS
SKY	SDBLS
SKY	SSGLB
SKY	SSGLS
STG	NSDBLB
STG	NSDBLS
STG	NSSGLB
STG	NSSGLS
STG	SDBLB
STG	SSGLB
STG	SSGLS
WIND	NSDBLB
WIND	NSDBLS
...	...

Note: UNION statements including DISTINCT treat NULL values as duplicates.

In UNION queries, the keyword NULL can be included in the column list of one or both of the queries, so that columns not represented in all of the queries in the statement are retained in the result set.

4.3 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).

4.3.1 Searching for NULL

The search condition

```
WHERE column = NULL
```

will not retrieve any rows since NULL is not equal to anything. 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).

Find the names of the persons who made the reservations for those customers who have not yet checked in to the Hotel Skyline.

"Not checked in" is represented by NULL in the CHECKIN column.

```
SELECT  RESERVED_FNAME, RESERVED_LNAME
FROM    BOOK_GUEST
WHERE   CHECKIN IS NULL
AND     HOTELCODE = (SELECT  HOTELCODE
                     FROM    HOTEL
                     WHERE   NAME = 'SKYLINE');
```

RESERVED_FNAME	RESERVED_LNAME
OMAR	CHAFIR
AGNETA	ERIKSSON
SVEN	LINDHOLM
HENRIK	PIHL
URBAN	FRANSSON

Find the names of the guests who have checked in to the Hotel Laponia.

```
SELECT  GUEST_FNAME, GUEST_LNAME
FROM    BOOK_GUEST
WHERE   CHECKIN IS NOT NULL
AND     HOTELCODE = (SELECT  HOTELCODE
                     FROM    HOTEL
                     WHERE   NAME = 'LAPONIA');
```

GUEST_FNAME	GUEST_LNAME
CHRISTOPHER	DATE
STEN	JOHANSEN
STEFAN	HANSEN
GUNNAR	ALVE
NILS	KRISTOFERSEN
LARS	HOLMER
KNUT	KULLMER
JUDITH	SMITH
ADOLF	SCHMIDT
LAILA	ZETTERBERG
MATS	HANSSON

4.3.2 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:

Where are hotels with rooms that are more expensive than those at the hotel Skyline (hotel code SKY)?

```
SELECT  NAME, CITY
FROM    HOTEL AS H, ROOM_PRICES AS RP
WHERE   H.HOTELCODE = RP.HOTELCODE
AND     PRICE > ALL (SELECT  PRICE
                     FROM    ROOM_PRICES
                     WHERE   HOTELCODE = 'SKY');
```

This query works as long as there are no NULL values in the PRICE column. But introduce a new room type at Skyline with an unknown price, and the query results in an empty set. Moreover, the reverse query (hotels that are cheaper than all rooms at Skyline) also results in an empty set. (A justification for this is that as long as one price at Skyline is unknown, it is impossible to say whether rooms at other hotels are more or less expensive than those at Skyline).

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. Thus, the query above can also be written as follows:

```
SELECT  NAME, CITY
FROM    HOTEL AS H, ROOM_PRICES AS RP
WHERE   H.HOTELCODE = RP.HOTELCODE
AND     NOT EXISTS (SELECT  *
                    FROM    ROOM_PRICES
                    WHERE   HOTELCODE = 'SKY'
                    AND     (   PRICE <= RP.PRICE
                             OR PRICE IS NULL
                             OR RP.PRICE IS NULL ));
```

This formulation may be read as “Find hotels where no room at Skyline is cheaper than or the same price as any room in the hotel in question, as long as no prices are unknown”. The explicit PRICE IS NULL clause tests that if either of the components of the comparison is NULL, then the subselect is not empty, NOT EXISTS is false, and no row is returned.

In general, a query of the form ($ stands for any comparison operator):

```
SELECT  column-list
FROM    table1
WHERE   column1 $ ALL (SELECT  column2
                       FROM    table2
                       WHERE   condition)
```

is equivalent to

```
SELECT   column-list
FROM     table1
WHERE    NOT EXISTS (SELECT   *
                     FROM     table2
                     WHERE    condition
                     AND ( NOT table1.column1 $ table2.column2
                           OR  table1.column1 IS NULL
                           OR  table2.column2 IS NULL ));
```

A similar example is:

Where are hotels with rooms that have unknown prices or that are more expensive than rooms ***with known prices*** *at hotel Skyline?*

```
SELECT   NAME, CITY
FROM     HOTEL H, ROOM_PRICES RP
WHERE    H.HOTELCODE = RP.HOTELCODE
AND      NOT EXISTS (SELECT   *
                     FROM     ROOM_PRICES
                     WHERE    HOTELCODE = 'SKY'
                     AND      PRICE <= RP.PRICE);
```

This query does not exclude the occurrence of the NULL indicator from the comparisons. If there is an unknown price, then the hotel concerned will be included in the result set - even if the unknown price is at Skyline itself. (Skyline might have a room that is more expensive than all rooms with known prices at Skyline).

Formulated with ALL, this query would be:

```
SELECT   NAME, CITY
FROM     HOTEL H, ROOM_PRICES RP
WHERE    H.HOTELCODE = RP.HOTELCODE
AND      PRICE > ALL (SELECT   PRICE
                      FROM     ROOM_PRICES
                      WHERE    HOTELCODE = 'SKY'
                      AND      PRICE IS NOT NULL);
```

It is clear from the examples above that 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 (such as the CHECKIN and CHECKOUT columns in the BOOK_GUEST table). In this way the risks of confusion with NULL handling are minimized.

4.4 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.

The query used in the analysis is:

List the total amount due for reservations above number 1347. Sort the result by guest name.

```
SELECT  G.RESERVATION, G.GUEST_LNAME, SUM(B.COST)
FROM    BOOK_GUEST G, BILL B
WHERE   G.RESERVATION = B.RESERVATION
GROUP BY G.RESERVATION, G.GUEST_LNAME
HAVING  G.RESERVATION > 1347
ORDER BY GUEST_LNAME;
```

RESERVATION	GUEST_LNAME	
1351	ALBERTSON	420
1359	ALVE	100
1356	ANDERSSON	200
1401	BLOM	500
1358	CODD	100
1353	FIMPLEY	790
1352	FRANCIS	-
1397	GRANKVIST	100
1349	HANSEN	70
1404	HANSSON	500
1413	HEDIN	300
1391	HESTMAN	420
1361	HOLLINGSWORTH	100
1364	HOLLSTEN	200
1379	HOLMER	300
1348	JOHANSEN	200
1367	JOHNSSON	-
1374	KARLSSON	600
1372	KRISTOFERSEN	-
1388	KULLMER	440
1396	LAHTINEN	340
1363	LE FEVRE	740
1393	LE FEVRE	400
1383	LIND	240
1381	LINDE	900
1386	LUNDBECK	395
1357	NILSSON	455
1385	NYQVIST	600
1369	OLSSON	140
1370	OLSSON	100
1382	PEREZ	1310
1384	PERSSON	720
1392	PERSSON	1350
1398	RYDELL	100
1368	SCHLAGER	-
1395	SCHMIDT	200
1405	SELLIN	320
1389	SMITH	100
...	...	...

1. Subselects at the lowest nesting level are evaluated first

The first step in evaluating a select is to resolve subselects from the lowest level up, and conceptually replace the subselect with the result set. (The example here does not use a nested select). When all subselects are resolved, a (possibly complicated) single-level SELECT statement remains.

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    BOOK_GUEST G, BILL B
```

The FROM clause in the example produces an intermediate result set which is the full cross product of the BOOK_GUEST table and the BILL table.

3. The WHERE clause selects rows from the intermediate set

The WHERE clause selects rows from the full cross product result set that meet the criteria specified.

```
WHERE G.RESERVATION = B.RESERVATION
```

In this example the WHERE clause selects only those result set rows where the value in the RESERVATION column from the BOOK_GUEST table is equal to that in the RESERVATION column from the BILL table.

4. The GROUP BY clause groups the remaining result set

```
GROUP BY G.RESERVATION, G.GUEST_LNAME
```

G.RESERVATION	G.GUEST_LNAME	B.RESERVATION	B.COST
1347	DATE	1347	100
1347	DATE	1347	40
1347	DATE	1347	40
1348	JOHANSEN	1348	120
1348	JOHANSEN	1348	40
1348	JOHANSEN	1348	40
1349	HANSEN	1349	70
...	...	...	...

5. The HAVING clause selects groups

```
HAVING G.RESERVATION > 1347
```

G.RESERVATION	G.GUEST	B.RESERVATION	B.COST
1348	JOHANSEN	1348	120
1348	JOHANSEN	1348	40
1348	JOHANSEN	1348	40
1349	HANSEN	1349	70
...	...	...	...

6. The SELECT list selects columns, evaluates any expressions in the SELECT list, and reduces groups to single rows if set functions are used

```
SELECT G.RESERVATION,
       G.GUEST_LNAME,
       SUM(B.COST)
```

G.RESERVATION	G.GUEST_LNAME	
1348	JOHANSEN	200
1349	HANSEN	70
...	...	...

7. The results of subselects joined by UNION are merged

This example does not include a UNION.

8. The final result is sorted according to the ORDER BY clause

```
ORDER BY GUEST_LNAME;
```

RESERVATION	GUEST_LNAME	
1349	HANSEN	70
1348	JOHANSEN	200
...	...	...

5 DATA MANIPULATION

The previous chapter described how to retrieve data from tables with SELECT. 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.

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 Section 5.5). In order to make a CALL you must have EXECUTE privilege on the procedure.

5.1 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 subselect (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:

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 column.

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.

5.1.1 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.

Insert the values 'SUTB' and 'SUITE WITH BATH' into the ROOMTYPE and DESCRIPTION columns respectively into the ROOMTYPES table.

```
INSERT INTO ROOMTYPES (ROOMTYPE,DESCRIPTION)
VALUES ('SUTB','SUITE WITH BATH');
```

inserts the row

ROOMTYPE	DESCRIPTION
SUTB	SUITE WITH BATH

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 ROOMTYPES
VALUES ('SUTB','SUITE WITH BATH');
```

You can also insert the result of an expression into a table:

```
INSERT INTO ROOM_PRICES
VALUES ('LAP', 'SUTB', CURRENT_DATE,
         CURRENT_DATE + INTERVAL '32' DAY, 500 + 40);
```

HOTELCODE	ROOMTYPE	FROM_DATE	TO_DATE	PRICE
LAP	SUTB	1997-08-22	1997-09-23	540

5.1.2 Inserting with a subselect

Values to be inserted can also be specified in the form of a subselect, i.e. fetched from another table in the database.

```
INSERT INTO ROOMSTATUS
            SELECT DISTINCT  ROOMNO, 'KEY OUT'
            FROM             BOOK_GUEST
            WHERE            CHECKIN  IS NOT NULL
            AND              CHECKOUT IS NULL;
```

The same table cannot be listed in the subselect's FROM clause that is listed in the INSERT INTO clause - data cannot be selected from a table for insertion into the same table.

Inserting the result of a subselect 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.

5.1.3 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 BOOKGUEST (ROOMNO,KEYCODE)
VALUES ('SKY123',NEXT_VALUE OF KEYCODES_SEQUENCE);

INSERT INTO BILL (CHARGE_PERIOD_NO,COST)
VALUES (CURRENT_VALUE OF CHARGE_PERIOD_NO_SEQUENCE,400);
```

5.1.4 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 EXCHANGE_RATE (CURRENCY,RATE)
VALUES ('XYZ',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 EXCHANGE_RATE (CURRENCY)
VALUES ('XYZ');
```

5.2 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 7.25.

```
UPDATE EXCHANGE_RATE
SET    RATE = 7.25
WHERE  CURRENCY = 'USD';
```

Add 20 to the 1997-08-08 to 1997-11-14 price of a no-smoking, single room with shower in the Hotel Laponia.

```
UPDATE ROOM_PRICES
SET    PRICE = PRICE + 20
WHERE  ROOMTYPE  = 'NSSGLS'
AND    FROM_DATE = DATE '1997-08-08'
AND    TO_DATE   = DATE '1997-11-14'
AND    HOTELCODE = (SELECT HOTELCODE
                    FROM   HOTEL
                    WHERE  NAME = 'LAPONIA');
```

When a subselect is used in the search condition, the table being updated may not be used in the subselect.

Primary key columns can be updated provided the table is stored in a databank with the TRANS or LOG option.

5.3 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 hotels in STOCKHOLM from the HOTEL table.

```
DELETE FROM HOTEL
WHERE CITY = 'STOCKHOLM';
```

Delete all rows from the HOTEL table.

```
DELETE FROM HOTEL;
```

Delete information for guests with the last name SVENSON from the BILL table.

```
DELETE FROM BILL
WHERE RESERVATION IN (SELECT RESERVATION
                      FROM BOOK_GUEST
                      WHERE TRIM(GUEST_LNAME) = 'SVENSON');
```

When a subselect is used in the search condition, the table from which rows are deleted may not be used in the subselect.

5.4 Calling procedures

In addition to using data manipulation statements directly, as just described, it is also possible to manipulate table data by invoking 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 beginning of Chapter 6 of the Mimer SQL Reference Manual for a definition) 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.

A procedure is invoked by using the CALL statement. In the case of a result set procedure, used in an embedded SQL context, the CALL statement is not used directly but is specified in a cursor declaration. An ident requires EXECUTE privilege on a procedure in order to call it.

In the CALL statement, the value-expressions or assignment targets specified for each of the procedure parameters must be of data type which is assignment-compatible (see Section 4.5 of the *Mimer SQL Reference Manual*) with the parameter data type.

See Chapter 6 of the Mimer SQL Reference Manual for full details of the CALL statement and Chapter 8 of the Mimer SQL Programmer's Manual for a general discussion of the PSM functionality supported in Mimer SQL.

Invoke the procedure called ALLOCATE_ROOM.

```
CALL ALLOCATE_ROOM(142,:room_no);
```

Declare a cursor which will be used when result-set data is fetched from the result set procedure called WAKE_UP.

```
DECLARE room_nos CURSOR
    FOR CALL WAKE_UP(:query_interval);
```

5.5 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: A view will **always** be updatable if an INSTEAD OF trigger exists on the view, regardless of the conditions previously mentioned. If **all** the INSTEAD OF triggers on the view are dropped, the view will revert to not updatable if one or more of these conditions are true.

6 MANAGING TRANSACTIONS

6.1 Transaction principles

A transaction is an environment where it is possible to COMMIT some or all of the operations performed within it, or to ensure that all of them fail.

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*, 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. 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 corruption which might arise when two users attempt to change the same information at the same time.

See the Mimer SQL Programmer’s Manual for a more detailed discussion of transaction handling and database security.

6.2 Logging

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.

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.

TRANS — All operations on the databank are performed under transaction control. No transactions are logged.

NULL — 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.

All important databanks should be defined with LOG option, so that valuable data is not lost by any system failure.

6.3 Handling transactions

Transaction control statements in Mimer SQL are:

```
COMMIT;
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 READ ONLY;
SET SESSION READ WRITE;
SET SESSION ISOLATION LEVEL SERIALIZABLE;
SET SESSION ISOLATION LEVEL REPEATABLE READ;
SET SESSION ISOLATION LEVEL READ COMMITTED;
SET SESSION ISOLATION LEVEL READ UNCOMMITTED;
SET SESSION DIAGNOSTICS SIZE;
START TRANSACTION;
```

The following SQL statements may **not** be used inside a transaction:

ENTER	SET DATABASE	SET TRANSACTION
LEAVE	SET SESSION	START TRANSACTION
SET DATABANK	SET SHADOW	

The following SQL statements may be used inside a transaction provided they are the **only** statement executed in that transaction:

ALTER	CREATE PROCEDURE	DROP MODULE
COMMENT	CREATE SCHEMA	DROP PROCEDURE
CREATE BACKUP	CREATE SHADOW	DROP SCHEMA
CREATE DATABANK	CREATE TABLE	DROP SHADOW
CREATE FUNCTION	CREATE TRIGGER	DROP TABLE
CREATE INCREMENTAL BACKUP	DROP DATABANK	DROP TRIGGER
CREATE INDEX	DROP FUNCTION	UPDATE STATISTICS
CREATE MODULE	DROP INDEX	

In addition, the following BSQL commands (see Chapter 9) may not be used inside a transaction:

EXIT	LOAD	UNLOAD

6.3.1 Transaction handling in BSQL

Normal Mimer SQL transaction handling behavior applies in 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 Section 6.2 of the Mimer SQL Programmer's Manual.

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.

6.3.2 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 Section 6.3.5). If this has not been used to set the default transaction read option, the default is READ WRITE.

6.3.3 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 Section 6.3.5). 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 the description of SET TRANSACTION in the Mimer SQL Reference Manual for a full description of the effects that are possible, or guaranteed never to occur, at each of the four isolation levels.

6.3.4 Exception diagnostics within transactions

The SET TRANSACTION DIAGNOSTICS SIZE option allows the size of the diagnostics area to be defined. A 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 (5 or whatever has been defined to be the default by using SET SESSION) applies unless an alternative is explicitly set before each transaction.

6.3.5 Default transaction options

The SET SESSION statement is provided so that default values for certain transaction control settings can be defined.

The transaction control settings defined by SET TRANSACTION READ (see Section 6.3.2) and SET TRANSACTION ISOLATION LEVEL (see Section 6.3.3) apply to the single next transaction to be started. If these statements are not used explicitly before each transaction, the default settings apply.

SET SESSION allows the default settings for SET TRANSACTION READ and SET TRANSACTION ISOLATION LEVEL to be defined.

7 DEFINING THE DATABASE

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.

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.

This chapter describes the SQL statements for creating and managing the database structure. Examples are based on the database listed in Appendix A. In addition, BSQL provides specific commands for listing and describing database objects (see Chapter 9).

7.1 Creating idents and schemas

Idents are authorized users of the system or groups of users defined for easier ident management (see Section 2.1.3).

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).

The case of the characters in an ident name can be made significant by enclosing the string in double quotes ("").

Passwords are composed of case-significant characters and must be entered exactly as they are defined.

The statement for creating idents has the general form:

```
CREATE IDENT username
AS ident-type
[USING 'password'];
```

Passwords are required for user and program idents but are not used for group idents. Passwords are optional for OS_USER idents: an OS_USER with a password may connect to Mimer SQL in the same way as any other user ident.

When a USER, OS_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.

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 for details).

Create a user ident HOTELADM with the password "Hoteladm" (schema HOTELADM will also be automatically created).

```
CREATE IDENT HOTELADM
AS USER
USING 'Hoteladm';
```

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 for the group ECONOMY_DEPT.

```
CREATE IDENT ECONOMY_DEPT
AS GROUP;
```

Create a schema called NEW_SCHEMA.

```
CREATE SCHEMA NEW_SCHEMA;
```

Create table Y in the schema called NEW_SCHEMA.

```
CREATE TABLE NEW_SCHEMA.Y (A INTEGER);
```

Create schema called SCHEMA_S which contains sequence Z.

```
CREATE SCHEMA SCHEMA_S
  CREATE UNIQUE SEQUENCE Z;
```

7.2 Creating databanks

The statement for creating a databank has the general form

```
CREATE DATABANK databank-name
       [OF initial-size PAGES]
       [IN 'filename']
       [WITH transaction-control OPTION];
```

- The CREATE DATABANK clause defines the databank name.
- The optional OF clause allocates a specified number of Mimer pages. This sets the initial size of the file, it will be dynamically extended as space is required. If the OF clause is omitted, an initial file size of 1000 Mimer pages is assumed.
- The optional IN clause defines the file where the databank is to be stored (the form of the filename specification is machine-specific). If the IN clause is omitted, the file is created in the database home directory with the same name as *databank-name*.
- The optional WITH clause defines the transaction handling and logging option (see Section 6.2). If the WITH clause is omitted, the TRANS option is assumed.

Create a databank called GUESTDB with the default parameters (i.e. with TRANS option, of size 1000 Mimer pages and stored in the file called "guestdb".

```
CREATE DATABANK GUESTDB;
```

Create the ROOMSDB databank with LOG option, allocate 200 Mimer pages for it, and store it in a file call "rooms.dbnk".

```
CREATE DATABANK ROOMSDB
       OF 200 PAGES
       IN 'ROOMS.DBNK'
       WITH LOG OPTION;
```

At this point, the databank is empty.

7.3 Creating sequences

A sequence returns a series of integer values which is defined by specifying an initial value, a maximum value, an increment and whether the sequence is to be unique 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 sequence can be used to provide the default value for a domain or a table column, etc.

A unique sequence will never return the same value twice.

Create a sequence that defines the following (repeating) series of values: 1, 4, 7, 10, 3, 6, 9, 2, 5, 8, 1, 4, 7, 10, 3, 6, 9, 2, 5, 8, 1, 4, 7, 10, 3, 6…

```
CREATE SEQUENCE SEQ_1 INITIAL_VALUE = 1 INCREMENT = 3 MAX_VALUE = 10;
```

Create a sequence that defines the following series of values: 1, 4, 7, 10, 3, 6, 9, 2, 5, 8.

```
CREATE UNIQUE SEQUENCE SEQ_2 INITIAL_VALUE = 1 INCREMENT = 3
                             MAX_VALUE = 10;
```

7.4 Creating domains

Domains are used as data types in column definitions when creating tables

- to assist in keeping the database consistent
- to limit the data (particular values or data type) accepted in the columns
- to define default values for columns

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.

7.4.1 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.

Define the default value '-ND-' ("not defined") for the domain ROOMTYPE.

```
CREATE DOMAIN ROOMTYPE
       AS CHAR(4)
       DEFAULT '-ND-';
```

Define the current user's name as the default value for the domain NAME.

```
CREATE DOMAIN NAME AS CHAR(128)
       DEFAULT CURRENT_USER;
```

Define the domain CHARGE_PERIOD_VALUE which uses the sequence CHARGE_PERIOD_NO_SEQUENCE to provide a default value.

```
CREATE DOMAIN CHARGE_PERIOD_VALUE AS INTEGER
       DEFAULT CURRENT_VALUE OF CHARGE_PERIOD_NO_SEQUENCE;
```

Domains defining default values can also include check clauses. You could define the ROOMTYPE domain as:

```
CREATE DOMAIN ROOMTYPE
              AS CHAR(4)
              DEFAULT '-ND-'
              CHECK (VALUE IS NOT NULL);
```

This means that the NULL indicator will not be accepted into columns belonging to this domain.

If the default value is defined as being outside the check constraint this ensues that an explicit value must always be inserted into the column.

7.4.2 Domains with a check clause

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 Section 5.10 of the Mimer SQL Reference Manual) 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.

The domain CALENDAR, created below, uses a check clause to limit the range of accepted values:

```
CREATE DOMAIN CALENDAR
       AS     DATE
       CHECK (VALUE BETWEEN DATE '1996-01-01' AND
                            DATE '2099-12-31');
```

7.5 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 (they are included in the examples here for clarity).

Create the table EXCHANGE_RATE with two columns. Name the first column CURRENCY, make it of the CHARACTER data type with a maximum of three characters. Name the second column RATE and make it of the data type DECIMAL with a total of six digits, three of which can be decimal values. Declare the CURRENCY column as the primary key and place this table in the HOTELDB databank.

```
CREATE TABLE EXCHANGE_RATE (CURRENCY CHAR(3) NOT NULL,
                            RATE     DECIMAL(6,3),
                            PRIMARY KEY (CURRENCY))
       IN HOTELDB;
```

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 (see the Mimer SQL Reference Manual for details of how the best databank is chosen).

The empty table now exists in the databank. Data is inserted into the table with the INSERT statement (see Section 5.1).

The preceding example shows the simplest form of column list. The following variants may also be used:

- columns belonging to domains
- 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 BOOK_GUEST 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 for a full description of the table creation facilities.

```
CREATE TABLE BOOK_GUEST (RESERVATION     INTEGER(5)     NOT NULL,
                         BOOKING_DATE    DATE
                           DEFAULT  CURRENT_DATE        NOT NULL,
                         HOTELCODE       HOTELCODE      NOT NULL,
                         ROOMTYPE        ROOMTYPE       NOT NULL,
                         COMPANY         VARCHAR(100)   NOT NULL,
                         TELEPHONE       CHAR(15),
                         RESERVED_FNAME  PERSONNAME,
                         RESERVED_LNAME  PERSONNAME,
                         ARRIVE          DATE           NOT NULL,
                         DEPART          DATE           NOT NULL,
                         GUEST_FNAME     PERSONNAME,
                         GUEST_LNAME     PERSONNAME,
                         ADDRESS         VARCHAR(50),
                         CHECKIN         DATE,
                         CHECKOUT        DATE,
                         ROOMNO          ROOMNO,
                         PAYMENT         CHAR(10),
       PRIMARY KEY (RESERVATION),
       FOREIGN KEY (HOTELCODE) REFERENCES HOTEL,
       FOREIGN KEY (ROOMTYPE)  REFERENCES ROOMTYPES,
       FOREIGN KEY (ROOMNO)    REFERENCES ROOMS ON DELETE NO ACTION,
       CHECK (ARRIVE < DEPART AND CHECKIN <= CHECKOUT))
       IN HOTELDB;
```

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:

```
CREATE TABLE BOOK_GUEST
             (RESERVATION  INTEGER(5) NOT NULL PRIMARY KEY,
              BOOKING_DATE DATE       DEFAULT CURRENT_DATE NOT NULL,
              HOTELCODE    HOTELCODE  NOT NULL REFERENCES HOTEL,
              ROOMTYPE     ROOMTYPE   NOT NULL REFERENCES ROOMTYPES,
              ...
```

7.5.1 Column definitions

Domains are used for many columns in the example database 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.

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. CHECKIN and CHECKOUT in the table BOOK_GUEST, where NULL indicates that the reservation has not checked in or out). The presence of NULL values can often complicate the formulation of queries (see Section 4.3). Take particular care to exclude NULL from numerical columns which are to be used for mathematical operations.

7.5.2 The primary key constraint

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.

7.5.3 Unique constraint

A unique constraint can 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.

This is the recommended way of defining a unique constraint, the other methods described below are mentioned for information only.

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.

Creating a UNIQUE index on the table has the same effect as a unique constraint.

7.5.4 Foreign keys - referential constraints

Use foreign keys to maintain integrity between the contents of related tables.

Note: The tables referenced in a foreign key clause of a table definition must exist prior to the definition of the foreign key (unless the key is in the reference table itself, to ensure referential integrity within a table or the table definition is within a create schema statement and the foreign key constraint refers to a table in the same schema definition statement).

The number of columns listed as FOREIGN KEY must be the same as the number of columns in the primary key of the REFERENCES table, unless columns in an unique constraint are referenced explicitly in a column list (see the CREATE TABLE syntax in the Mimer SQL Reference Manual for details). The nth FOREIGN KEY column corresponds to the nth column in the primary key of the REFERENCES table, and the data types and lengths of corresponding columns must be identical. Columns may not be used more than once in the same FOREIGN KEY clause.

If the NULL indicator is permitted in a foreign key, then either at least one of the columns in the foreign key is NULL or the values in the foreign key columns must be present in the corresponding primary key columns of the reference table.

A table definition may contain as many FOREIGN KEY references as required. The same column in the table may be used in separate FOREIGN KEY clauses referring to different REFERENCES tables.

Note: A table containing a foreign key reference or referenced in a foreign key must be stored in a databank with either the TRANS or LOG option.

The BOOK_GUEST table has three foreign key references:

```
CREATE TABLE BOOK_GUEST (RESERVATION    INTEGER(5),
                         BOOKING_DATE   DATE
                           DEFAULT  CURRENT_DATE     NOT NULL,
                         HOTELCODE      HOTELCODE    NOT NULL,
                         ROOMTYPE       ROOMTYPE     NOT NULL,
                            .
                            .
                         ROOMNO         ROOMNO,
                            .
        FOREIGN KEY (HOTELCODE) REFERENCES HOTEL,
        FOREIGN KEY (ROOMTYPE)  REFERENCES ROOMTYPES,
        FOREIGN KEY (ROOMNO)    REFERENCES ROOMS ON DELETE NO ACTION    )
                            .
                            .
```

These maintain referential integrity as follows:

- FOREIGN KEY (HOTELCODE) REFERENCES HOTEL
  Data that is not present in the HOTELCODE column of the HOTEL table will not be accepted in the HOTELCODE column in the BOOK_GUEST table.
- FOREIGN KEY (ROOMTYPE) REFERENCES ROOMTYPES
  Data that is not present in the ROOMTYPE column of the ROOMTYPES table will not be accepted in the ROOMTYPE column in the BOOK_GUEST table.
- FOREIGN KEY (ROOMNO) REFERENCES ROOMS
  Data that is not present in the ROOMNO column of the ROOMS table will not be accepted in the ROOMNO column in the BOOK_GUEST table.

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

7.5.5 Check constraints

Check constraints in table definitions are used to make sure that data in a column in the table fits certain conditions. This section gives three different examples of check constraints.

Note that the first two examples shown below are not used in the example database.

Limit the city for hotels to Stockholm or Gothenburg.

```
CREATE TABLE HOTEL (HOTELCODE HOTELCODE,
                    NAME      CHAR(15)  NOT NULL,
                    CITY      CHAR(15)  NOT NULL,
                    OVERBOOK  BOOK_RATE NOT NULL,
       PRIMARY KEY (HOTELCODE),
       CONSTRAINT CITY_CHECK CHECK (CITY IN ('STOCKHOLM','GOTHENBURG')))
       IN HOTELDB;
```

Prevent blank entries in the HOTELCODE column.

```
CREATE TABLE HOTEL (HOTELCODE HOTELCODE,
                    NAME      CHAR(15)  NOT NULL,
                    CITY      CHAR(15)  NOT NULL,
                    OVERBOOK  BOOK_RATE NOT NULL,
       PRIMARY KEY (HOTELCODE),
       CHECK (HOTELCODE <> ' '))
       IN HOTELDB;
```

This check clause extends any limitations imposed by the HOTELCODE domain definition. The extension applies only to this table, and does not affect other columns in the database which belong to the HOTELCODE domain. The constraint name, CITY_CHECK in the first example above, 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 in this manual.

Make sure that arrival dates are before departure dates.

```
CREATE TABLE BOOK_GUEST (    .
                             .
                ARRIVE          DATE           NOT NULL,
                DEPART          DATE           NOT NULL,
                             .
                             .
                CHECKIN         DATE,
                CHECKOUT        DATE,
                             .
                             .
        CHECK (ARRIVE < DEPART AND CHECKIN <= CHECKOUT))
        IN HOTELDB;
```

Check conditions allow any value that does not evaluate to false in the check condition. This means that unknown values (the NULL indicator) are allowed in columns restricted by the check condition. Thus the check condition above does not exclude NULL from the CHECKIN and CHECKOUT columns (NULL values give an unknown result in the condition).

7.6 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 routines.

Triggers contain the same constructs as routines but are created on tables or views (depending on the type of trigger) and execute instead of, before or after a specified data manipulation operation.

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.

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.

The CREATE TRIGGER statement is used to define a trigger on a table or view.

Refer to the Mimer SQL Reference Manual for the syntax definitions for CREATE FUNCTION, CREATE MODULE, CREATE PROCEDURE and CREATE TRIGGER, and Chapter 8 of the Mimer SQL Programmer's Manual for a general discussion of the PSM functionality in Mimer SQL.

Note: The examples that follow show 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. This is the case for many of the SQL/PSM statements. See Section 9.1 for details about running BSQL. 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(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 X INTEGER, OUT Y 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 X INTEGER)
  MODIFIES SQL DATA
  BEGIN
    ...
  END;
  DECLARE FUNCTION FUNC_1() RETURNS INTEGER
  READS SQL DATA
  BEGIN
    ...
  END;
END MODULE
@
```

Create a trigger which will execute before UPDATE operations on table BOOK_GUEST.

```
@
CREATE TRIGGER VERIFY_GUEST_UPDATES BEFORE UPDATE ON BOOK_GUEST
  BEGIN
    ...
  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 Section 8.9 of 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.

7.7 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 Section 5.5 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.

The example database does not contain any view definitions. Two examples are given below:

Create a restriction view of the BOOK_GUEST table called RECEPTION containing limited information for the hotel reception (reservation number, customer name, check-in date and room number).

```
CREATE VIEW RECEPTION (RESERVATION, FNAME, LNAME, DATE, ROOM)
       AS   SELECT RESERVATION, GUEST_FNAME, GUEST_LNAME,
            CHECKIN, ROOMNO
       FROM BOOK_GUEST;
```

RESERVATION	FNAME	LNAME	DATE	ROOM
1348	STEN	JOHANSEN	1997-08-23	LAP205
1349	STEFAN	HANSEN	1997-08-23	LAP206
1350	SALLY	WEBERT	1997-08-06	SKY124
1351	ANNA	ALBERTSON	1997-08-06	SKY125
1352	MARK	FRANCIS	1997-08-14	WINS103
1353	ALFRED	FIMPLEY	1997-09-03	SKY110
...	...		...	...
...	...		...	...

Create a join view listing the billing details for each reservation.

```
CREATE VIEW CHARGE_DESCRIPTION
    AS SELECT  RESERVATION, COST, DESCRIPTION
       FROM    BILL, CHARGES
       WHERE   BILL.CHARGE_CODE = CHARGES.CHARGE_CODE;
```

If the view definition does not include a list of column names, the columns in the view will be named after the columns listed in the SELECT clause.

RESERVATION	COST	DESCRIPTION
1348	100	LODGING
...	...	...

7.7.1 Check options

Check options 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 GUEST_VIEW
    AS SELECT  RESERVATION, HOTELCODE, GUEST_FNAME, GUEST_LNAME,
                CHECKIN, ROOMNO
       FROM    BOOK_GUEST
       WHERE   HOTELCODE = 'STG' OR HOTELCODE = 'WINS'
       WITH    CHECK OPTION;
```

RESERVATION	HOTELCODE	GUEST_FNAME	GUEST_LNAME	CHECKIN	ROOMNO
1355	STG	INGER	SVENSON	1997-09-01	STG111
1363	WINS	PAULE	LE FEVRE	1997-08-20	WINS117
1364	STG	LARS	HOLLSTEN	1997-09-01	STG116
1367	WINS	EARNST	JOHNSSON	1997-09-06	WINS109
1371	STG	MARY	TENMAR	1997-08-29	STG010
1382	WINS	JULIO	PEREZ	1997-09-29	WINS119
1383	STG	ROBERT	LIND	1997-08-31	STG142
1384	WINS	SIGWARD	PERSSON	1997-09-25	WINS120
1385	WINS	RUNE	NYQVIST	1997-09-25	WINS121
1398	STG	LENNART	RYDELL	1997-09-30	STG1421
1401	STG	JAN	BLOM	1997-09-23	STG001
1408	STG	EINAR	SUNDMAN	1997-09-20	STG117
1412	WINS	JOHAN	TORP	1997-09-30	WINS119

The check option in the view definition (WITH CHECK OPTION) means that no new rows may be inserted into the view if the value for the HOTELCODE column is not STG or WINS. 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 NEW_VIEW
    AS SELECT  RESERVATION, HOTELCODE, GUEST_FNAME, GUEST_LNAME
       FROM    GUEST_VIEW
       WHERE   RESERVATION > 1385;
```

7.8 Creating secondary indexes

Secondary indexes are maintained by the system and are invisible to the user. The index is automatically used during searching when it improves the efficiency of the search.

Any column(s) may be specified as a secondary index. 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. Index names can be made up of a maximum of 128 characters.

If, for instance, you want to know which room a certain person is staying in at a hotel, Mimer SQL would have to search successively through the customer reference numbers and the names corresponding to each in order to find the information you want. If, however, you create a secondary index on guest names, Mimer SQL would search for the name of that person directly in the secondary index, which would save time.

Create a secondary index called NAME on the GUEST_LNAME column in the BOOK_GUEST table.

```
CREATE INDEX NAME
ON BOOK_GUEST (GUEST_LNAME);
```

Primary key columns may also be included in a secondary index. If a table has the primary key "A,B,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", "A,B", and "A,B,C". 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).

Create a UNIQUE secondary index called OCCUPANCY on the GUEST_LNAME and ROOMNO columns in the BOOK_GUEST table.

```
CREATE UNIQUE INDEX OCCUPANCY
ON BOOK_GUEST (GUEST_LNAME, ROOMNO);
```

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.

Secondary indexes can improve the efficiency of data retrieval; but does introduce 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.

7.9 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. Thus the table ROOMS in the schema HOTELADM has the qualified name:

```
HOTELADM.ROOMS
```

The ident called HOTELADM need only refer to it as:

```
ROOMS
```

If another user should wish to use this table, he must refer to it by its fully qualified name since he does not have the same name as the schema to which the table belongs.

If a user named James, who wishes to refer to the ROOMS table, belonging to the schema HOTELADM, as simply ROOMS, 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 ROOMS
FOR HOTELADM.ROOMS;
```

Another user can then create his own synonym for the ROOMS synonym which now exists in schema “James”, which has the fully qualified name:

```
JAMES.ROOMS
```

Synonyms are particularly useful when several users refer to a common table, such as HOTELADM.ROOMS, HOTELADM.HOTEL, etc. With synonyms, several users can work in the same apparent environment without needing to refer to the tables by their qualified names.

7.10 Commenting objects

Comments may be stored against any of the following objects:

```
COLUMN      FUNCTION    MODULE      SEQUENCE    TABLE
DATABANK    IDENT       PROCEDURE   SHADOW      TRIGGER
DOMAIN      INDEX       SCHEMA      SYNONYM     VIEW
```

Store the comment “MIMER Hotels Databank” on the HOTELDB databank.

```
COMMENT ON DATABANK HOTELDB IS 'MIMER Hotels Databank';
```

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 schema to which the object belongs may store a comment for the object.

Comments are for information only and do not affect data retrieval or manipulation in any way. Comments may be read with the DESCRIBE command (Chapter 9) or by retrieving the appropriate columns from the INFORMATION_SCHEMA views (see Section 7.1 of the Mimer SQL Reference Manual).

7.11 Altering databanks, tables and idents

7.11.1 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 increase the file size allocated for the databank

Change which file the HOTELDB is stored in from its previous file to file "SQLDB:HOTELDB.DBF" (the file specification is in Alpha/Open VMS format).

```
ALTER DATABANK HOTELDB
 INTO 'SQLDB:HOTELDB.DBF';
```

Note: This statement changes the file name 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 ... INTO to change the file specification in the data dictionary.

Change the option on the HOTELDB databank from TRANS to LOG.

```
ALTER  DATABANK HOTELDB
   TO  LOG OPTION;
```

Increase the size of the HOTELDB database by 20 Mimer pages.

```
ALTER  DATABANK HOTELDB
  ADD  20 PAGES;
```

Note: Use of the ALTER DATABANK ... ADD statement is not strictly necessary. However, increasing the file allocation by a relatively large figure can help to minimize file fragmentation and improve response times.

7.11.2 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 change a column in an existing table to have a specified data type or to belong to a specified domain, provided the old and new data types are assignment-compatible (see Section 4.5 of the *Mimer SQL Reference Manual*) and the column is not be referenced by any constraints or views
- to drop the default value for a column in an existing table

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 indicator.

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.

Add a column called NOSMOKE with a data type of CHAR(1) to the BOOK_GUEST table.

```
ALTER  TABLE BOOK_GUEST
ADD    NOSMOKE CHAR(1);
```

This creates a column containing the NULL indicator for each row in the table. If an constraint is added to a table it is checked that the data in the table fulfil the restriction in the constraint.

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, depending 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.

Drop the column TELEPHONE from the table BOOK_GUEST, subject to the condition that there are no other objects dependent on this column.

```
ALTER TABLE BOOK_GUEST DROP TELEPHONE RESTRICT;
```

Change the length of the column ADDRESS in the table BOOK_GUEST

```
ALTER TABLE BOOK_GUEST ALTER COLUMN ADDRESS VARCHAR(100);
```

Drop the column TELEPHONE from the table BOOK_GUEST, if dependent objects exist, these are dropped as well.

```
ALTER TABLE BOOK_GUEST DROP TELEPHONE CASCADE;
```

Change the default value for the column BOOKING_DATE, the new default value is current date.

```
ALTER TABLE BOOK_GUEST ALTER BOOKING_DATE SET DEFAULT CURRENT_DATE;
```

Drop the check constraint CITY_CHECK from the HOTEL table.

```
ALTER TABLE HOTEL DROP CONSTRAINT CITY_CHECK;
```

Redefine a foreign key constraint for the BOOK_GUEST table.

```
ALTER TABLE BOOK_GUEST DROP CONSTRAINT SQL_FOREIGN_KEY_4375;
ALTER TABLE BOOK_GUEST ADD CONSTRAINT ROOMS_FOREIGN FOREIGN KEY(ROOMNO)
REFERENCES ROOMS ON DELETE CASCADE;
```

Drop the default value for the column BOOKING_DATE.

```
ALTER TABLE BOOK_GUEST ALTER BOOKING_DATE DROP DEFAULT;
```

7.11.3 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.

Change the user SAMMY's password to '*SamJo*'.

```
ALTER IDENT SAMMY
USING 'SamJo';
```

7.11.4 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.

7.12 Dropping objects from the database

The DROP statement is used to drop the following objects from the database:

```
DATABANK    IDENT       PROCEDURE    SHADOW       TRIGGER
DOMAIN      INDEX       SCHEMA       SYNONYM      VIEW
FUNCTION    MODULE      SEQUENCE     TABLE
```

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 Section 7.11.2).

7.12.1 Dropping databanks and tables

Drop the HOTEL table.

```
DROP TABLE HOTEL RESTRICT;
```

If the keyword CASCADE is specified, all views, synonyms and indexes based on HOTEL are also dropped as well as any functions, procedures and triggers referencing the table.

Drop the HOTELDB databank.

```
DROP DATABANK HOTELDB RESTRICT;
```

If the keyword CASCADE is specified, all tables in the HOTELDB 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.

7.12.2 Dropping sequences

When a sequence is dropped, all the objects (i.e. constraints, domains, functions, procedures, tables, triggers and views) referencing the sequence are also dropped.

Drop the SEQ_1 sequence.

```
DROP SEQUENCE SEQ_1 CASCADE;
```

The specification of CASCADE ensures that the sequence is dropped even if it is being referenced by other objects in the database.

7.12.3 Dropping domains

When a domain is dropped, existing columns assigned the domain retain all the properties of the domain. No new columns may however be assigned the domain.

Drop the BOOK_RATE domain.

```
DROP DOMAIN BOOK_RATE RESTRICT;
```

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.

7.12.4 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 GUEST_CONNECT ident.

```
DROP IDENT GUEST_CONNECT RESTRICT;
```

7.12.5 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 modules, routines and triggers be created by running a **command file** so they can be easily reconstructed in case of being dropped in error.

Drop the function called BILL_TOTAL.

```
DROP FUNCTION BILL_TOTAL CASCADE;
```

Drop the procedure called ADD_LODGING.

```
DROP PROCEDURE ADD_LODGING CASCADE;
```

Drop the module called ROOMS_ADMIN.

```
DROP MODULE ROOMS_ADMIN CASCADE;
```

Drop the trigger called VERIFY_GUEST_UPDATES.

```
DROP TRIGGER VERIFY_GUEST_UPDATES CASCADE;
```

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 (this is not a cascade effect, but it may provoke cascade effects).

- 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).

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.

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, the creator of a program ident may enter it, the creator of a table has full access privileges, the creator of 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.

8.1 Ident hierarchy

In the initial installation, one user ident, the system administrator with user ident name SYSADM, is automatically created. The system administrator has BACKUP, DATABANK, IDENT, SCHEMA, SHADOW and STATISTICS privileges, with GRANT OPTION, and SELECT access on all tables and views in the data dictionary, also with GRANT OPTION. 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 systems) that the system administrator is prevented from accessing the actual contents of the database; the administrator's job is concerned with managing objects in the system, not with the data.

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.

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 or OS_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. OS_USER idents are allowed access to the database on the authorization of a valid log-in to the operating system. For maximum protection, do not use OS_USER idents.

- 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 and OS_USER idents should not in general be granted privileges to create objects (i.e. granted DATABANK, IDENT, SCHEMA, SHADOW or TABLE privileges). In this way, individual user idents may be dropped with no cascading effects except loss of views created by the user.

- 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.

8.2 Granting privileges

8.2.1 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:

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.

Give the ident HOTELADM the privilege to create new databanks.

```
GRANT  DATABANK
   TO  HOTELADM;
```

Give the idents AUDIT and ECONOMY_DEPT the privilege to create new idents with grant option.

```
GRANT  IDENT
   TO  AUDIT, ECONOMY_DEPT
 WITH  GRANT OPTION;
```

8.2.2 Granting object privileges

Object privileges are held by idents on database objects (functions, procedures, programs, groups, tables, domains and sequences). The four object privileges are:

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.
USAGE	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.

Give STEVE and MARIANNE the privilege to execute the SUMMARY_STATS procedure.

```
GRANT  EXECUTE ON PROCEDURE SUMMARY_STATS
   TO  STEVE, MARIANNE;
```

Give ECONOMY_DEPT the privilege to enter the AUDIT program ident.

```
GRANT  EXECUTE ON PROGRAM AUDIT
   TO  ECONOMY_DEPT;
```

Make STEVE, MARIANNE and JAMES members of the ECONOMY_DEPT group with grant option.

```
GRANT  MEMBER ON ECONOMY_DEPT
   TO  STEVE, MARIANNE, JAMES
 WITH  GRANT OPTION;
```

Give the members of the ECONOMY_DEPT group the privilege to create new tables in the HOTELDB databank.

```
GRANT  TABLE ON HOTELDB
   TO  ECONOMY_DEPT;
```

Give the members of the ECONOMY_DEPT group the privilege to use the LOCAL_CURRENCY domain.

```
GRANT  USAGE ON DOMAIN LOCAL_CURRENCY
   TO  ECONOMY_DEPT;
```

8.2.3 Granting access privileges

Access privileges define what data the idents are allowed to manipulate in tables. There are five access privileges:

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.
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 (ALL may be followed by the optional keyword PRIVILEGES).

Give JAMES the privilege to read, insert, and delete rows from the BOOK_GUEST table and give the ident the right to pass these privileges on to other idents.

```
GRANT   SELECT, INSERT, DELETE
   ON   BOOK_GUEST
   TO   JAMES
 WITH   GRANT OPTION;
```

Give ECONOMY_DEPT and AUDIT all privileges that you hold on the table CHARGES but do not give them the right to pass these privileges on to other idents.

```
GRANT   ALL ON CHARGES
   TO   ECONOMY_DEPT, AUDIT;
```

Give ECONOMY_DEPT the privilege to update all columns in the BOOK_GUEST table.

```
GRANT   UPDATE ON BOOK_GUEST
   TO   ECONOMY_DEPT;
```

Give RECEPTION the privilege to update only the GUEST_LNAME, ADDRESS, and ROOMNO columns in the BOOK_GUEST table.

```
GRANT   UPDATE (GUEST_LNAME,ADDRESS,ROOMNO)
   ON   BOOK_GUEST
   TO   RECEPTION;
```

Give ECONOMY_DEPT the right to use the ROOMS table as a foreign key.

```
GRANT   REFERENCES
   ON   HOTELADM.ROOMS
   TO   ECONOMY_DEPT;
```

8.3 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 Section 8.3.4 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, but he still has the privilege.

8.3.1 Revoking system privileges

Take away the privilege to create new databanks from the ident HOTELADM.

```
REVOKE  DATABANK
  FROM  HOTELADM RESTRICT;
```

Take away the privilege to create new idents from the idents AUDIT and ECONOMY_DEPT.

```
REVOKE  IDENT
  FROM  AUDIT, ECONOMY_DEPT RESTRICT;
```

Revoking system privileges does not affect objects already created under the authorization of the privilege.

8.3.2 Revoking object privileges

Take away the privilege to execute the ALLOCATE_ROOM procedure from STEVE and MARIANNE.

```
REVOKE  EXECUTE ON PROCEDURE ALLOCATE_ROOM
  FROM  STEVE, MARIANNE RESTRICT;
```

Take away the privilege to enter the AUDIT program from the ident ECONOMY_DEPT.

```
REVOKE  EXECUTE ON PROGRAM AUDIT
  FROM  ECONOMY_DEPT RESTRICT;
```

Take away the idents' STEVE, MARIANNE and JAMES memberships in the group ECONOMY_DEPT.

```
REVOKE  MEMBER ON ECONOMY_DEPT
  FROM  STEVE, MARIANNE, JAMES RESTRICT;
```

Take away the right to use the domain BOOK_RATE from the ident ECONOMY_DEPT.

```
REVOKE  USAGE ON DOMAIN BOOK_RATE
  FROM  ECONOMY_DEPT RESTRICT;
```

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.

8.3.3 Revoking access privileges

Revoke the privileges to delete and insert rows and to retrieve data from the BOOK_GUEST table from the ident MARIANNE.

```
REVOKE  SELECT, DELETE, INSERT ON BOOK_GUEST
  FROM  MARIANNE RESTRICT;
```

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 ROOMS as foreign keys from ECONOMY_DEPT.

```
REVOKE  REFERENCES
    ON  ROOMS
  FROM  ECONOMY_DEPT RESTRICT;
```

Revoke the right to grant select on the BOOK_GUEST table from JAMES. Any grants that JAMES has made will also be revoked.

```
REVOKE  GRANT OPTION FOR SELECT
    ON  BOOK_GUEST
  FROM  JAMES CASCADE;
```

The keyword ALL may be used as a shorthand for all the privileges that may be revoked in the current context.

8.3.4 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. All procedures, functions and triggers that reference objects accessed by the WITH GRANT OPTION 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)

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 (i.e. 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.

9 BSQL COMMANDS

BSQL is a facility for executing SQL statements in batch jobs. All SQL statements may be used in BSQL. This chapter documents the set of specific batch-oriented commands.

9.1 Running BSQL

BSQL can be run from a batch job or from a terminal. Operation from a terminal can be used to execute statements entered directly or written in sequential files.

It is only possible to specify up to 80 characters on the command line in BSQL. Input lines taken from a sequential file can be longer than 80 characters.

Note: The “@” character should be used to delimit a complex SQL statement where the normal end-of-statement character “;” appears before the end of the statement (e.g. CREATE FUNCTION, CREATE PROCEDURE, CREATE TRIGGER). It is also useful to use in conjunction with large statements, e.g. create schema, in which case the error reporting in BSQL will give more information about where the error occurred. The use of “@” cannot be used for grouping a number of “simple” SQL statements so that they execute as one single statement, but it is provided to give the SQL interpreter advance warning that a complex SQL statement appears between the “@” characters which contains end-of-statement markers occurring before the true end of construct.

9.1.1 Running BSQL from a batch job

To run BSQL unattended from a batch job, create a batch file with the following contents:

- command to start BSQL
- username
- password
- SQL statements and BSQL commands
- EXIT command (or end of file)

Note: For unattended operation, a batch file must either include the Mimer SQL ident username and password in explicit form or connect as OS_USER. For security reasons, make sure that your batch 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 a BSQL terminal session.

9.1.2 Running BSQL via the terminal

For instructions on how to start BSQL see Section 3.8 of the Mimer SQL System Management Handbook. Starting BSQL displays the following screen:

```
   MMMMM     MMMMM  MMMMM  MMMMM     MMMMM  MMMMMMMMMM  MMMMMMMM
   MMMMMM   MMMMMM  MMMMM  MMMMMM   MMMMMM  MMMMMMMMMM  MMMMMMMMM
    MMMMMM MMMMMM    MMM    MMMMMM MMMMMM    MMM   MMM   MMM  MMM
    MMMMMMMMMMMMM    MMM    MMMMMMMMMMMMM    MMMMM       MMMMMMM
    MMM MMMMM MMM    MMM    MMM MMMMM MMM    MMM   MMM   MMM  MMM
   MMMM  MMM  MMMM  MMMMM  MMMM  MMM  MMMM  MMMMMMMMMM  MMMM  MMMM
   MMMM   M   MMMM  MMMMM  MMMM   M   MMMM  MMMMMMMMMM  MMMM  MMMM

(C) Copyright Mimer Information Technology AB. All rights reserved.

                          M I M E R / B S Q L
                             Version 8.2.1

                           Username:
                           Password:
```

When the username and correct password are entered, the BSQL prompt will be shown:

```
SQL>
```

BSQL commands and SQL statements can now be entered. Output will be echoed on the terminal.

9.1.3 BSQL command line editing

Unix

Command line editing is available in the BSQL program, which uses a line-oriented interface. The following functions are available:

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
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

The arrow keys can be used 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 for the terminal that, if they overlap, override those described above. The terminal settings can be listed using the Unix **stty -a** command.

9.2 BSQL commands

Command	Function
CLOSE	Closes active log files
DESCRIBE	Describes a specified object
EXIT	Leaves BSQL
LIST	Lists information on a specified object
LOAD	Loads data into a table
LOG	Logs input, output or both on a sequential file
READ INPUT	Reads commands from a sequential file
SET ECHO	Specifies whether lines are echoed to the terminal during READ INPUT
SET LINECOUNT	Sets the terminal page size
SET LINESPACE	Sets the number of blank lines between each output record
SET LINEWIDTH	Sets the terminal page width
SET LOG	Stops or resumes logging input, output or both
SET MESSAGE	Specifies whether messages are displayed on the terminal
SET OUTPUT	Specifies whether output should be written to the terminal
SET PAGELENGTH	Defines the page length of output file
SET PAGEWIDTH	Defines the page width of output file
SHOW SETTINGS	Displays current values of all set options
UNLOAD	Unloads data from a table
WHENEVER	Sets action to be taken in response to an error or warning

BSQL commands are not case sensitive.

Note on syntax descriptions

In the syntax descriptions, items in square brackets ([]) are optional. Items separated by a vertical bar (|) are alternatives. For example:

READ [COMMAND | ALL] [INPUT FROM] 'filename';

allows the following forms

READ COMMAND INPUT FROM 'filename';

READ ALL INPUT FROM 'filename';

READ INPUT FROM 'filename';

READ 'filename';

CLOSE

Closes log files.

Syntax

CLOSE [INPUT|OUTPUT|INPUT,OUTPUT] log;

Description

The command closes the specified log file. If no log file is specified, all active log files are closed.

DESCRIBE

Describes a specified object.

Syntax

DESCRIBE [object-type [object-name]];

Description

The DESCRIBE command presents the following menu:

```
                 Menu for describe

1. Databank        6. Table        11. Trigger
2. Domain          7. View         12. Sequence
3. Ident           8. Module       13. Schema
4. Index           9. Procedure
5. Synonym        10. Function      0. Exit


 Select :_
```

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 can be provided in a batch file for unattended operation. However, DESCRIBE is most useful in interactive mode from a terminal.

DESCRIBE	OPTION	RESULT
DATABANK	BRIEF	Lists the following information on the specified databank: creator file space used allocated size physical file name option tables.
	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 physical file name option tables idents with table privilege comment creation date.
DOMAIN	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	OPTION	RESULT
IDENT	BRIEF	Lists the following information on the specified ident: creator ident type 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.
	FULL	Lists the following information on the specified ident: creator ident type accessible objects created objects comment creation date.
INDEX	BRIEF	Lists the following information on the specified index: table name and columns on which the index is defined sort order uniqueness comment creation date.
SYNONYM	BRIEF	Lists the following information on the specified synonym: schema and name of referenced table/view comment creation date.

DESCRIBE	OPTION	RESULT
TABLE	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.
	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 comment creation date date when statistics were generated.
VIEW	BRIEF	Lists the following information on the specified view: view definition comment creation date.
MODULE	BRIEF	List the following information on the specified module: module definition comment creation date.

DESCRIBE	**OPTION**	**RESULT**
PROCEDURE	BRIEF	Lists the following information on the specified procedure: parameters result items procedure attributes specific name.
	BY ACCESS	Lists the following information on the specified procedure: idents with execute privilege.
	BY REFERENCES	Lists the following information on the specified procedure: referencing objects referenced objects.
	FULL	Lists the following information on the specified procedure: parameters result items procedure attributes idents with execute privilege referencing objects referenced objects source definition module name specific name comment creation date.
FUNCTION	BRIEF	Lists the following information on the specified function: parameters result data type function attributes specific name.
	BY ACCESS	Lists the following information on the specified function: idents with execute privilege.
	BY REFERENCES	Lists the following information on the specified procedure: referencing objects referenced objects.
	FULL	Lists the following information on the specified function: parameters result data type function attributes specific name idents with execute privilege referencing objects referenced objects source definition module name comment creation date.

DESCRIBE	OPTION	RESULT
TRIGGER	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.
SEQUENCE	BRIEF	List the following information about the specified sequence: initial value increment value maximum value.
	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 referencing objects idents with usage privilege comment creation date.
SCHEMA	BRIEF	List the following information about the specified schema: schema owner contained objects comment creation date.

EXIT

Leave BSQL.

Syntax

EXIT;

Description

Terminates the BSQL session.

LIST

Lists information on a specified object.

Syntax

LIST [object-type];

Description

The LIST command presents the following menu:

```
                  Menu for List

1. Databanks      6. Synonyms     11. Functions
2. Domains        7. Tables       12. Triggers
3. Idents         8. Views        13. Sequences
4. Indexes        9. Modules      14. Schemata
5. Objects       10. Procedures    0. Exit

Select :_
```

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 can be provided in a batch file for unattended operation. However, LIST is most useful in interactive mode from a terminal.

LIST	**OPTION**	**RESULT**
DATABANKS	ALL	Lists all databanks in the database.
	CREATED BY	Lists databanks created by a specified ident.
	ALL SHADOWS	Lists all shadows in the database.

LIST	OPTION	RESULT
DOMAINS	ALL	Lists all domains in the database.
	CREATED BY	Lists domains created by a specified ident.
IDENTS	ALL	Lists all idents in the database.
	CREATED BY	Lists idents created by a specified ident.
INDEXES	ALL	Lists the secondary indexes in the database.
	CREATED BY	Lists secondary indexes created by a specified ident.
OBJECTS	ALL	Lists objects in the database.
	CREATED BY	Lists objects created by a specified ident.
	BY TYPE	Lists objects of a specified type.
SYNONYMS	ALL	Lists synonyms in the database.
	CREATED BY	Lists synonyms created by a specified ident.
TABLES	ALL	Lists tables in the database.
	CREATED BY	Lists tables created by a specified ident.
VIEWS	ALL	Lists views in the database.
	BY CREATOR	Lists views created by a specified ident.
MODULES	ALL	Lists all the modules in the database that are visible to (i.e. created by) the current ident.
PROCEDURES	ALL	Lists all the procedures the current ident has execute privilege on.
	CREATED BY	Lists procedures created by the specified ident.
FUNCTIONS	ALL	Lists all the functions the current ident has execute privilege on.
	CREATED BY	Lists functions created by the specified ident.
TRIGGERS	ALL	List triggers defined on tables accessible to current user
	CREATED BY	Lists procedures created by the specified ident.
SEQUENCES	ALL	Lists all the sequences the current ident has usage privilege on.
	CREATED BY	Lists sequences created by the specified ident.
SCHEMATA	ALL	List schemata created by the current ident

LOAD

The LOAD command can be used to load data from a sequential file into a target table.

Syntax

LOAD FROM 'file-name' INTO table-name <NULL | NONULL,> <DUPLICATES | NODUPLICATES> <LOGFILE 'file-name'> < (col-name POS(s:e), ..., col-name POS(s:e)) | DELIMITER 'character' >;

Description

NULL (default) specifies that the first byte for each column value in the input file is used to indicate whether the value is NULL or not. An ampersand (&) in this byte indicates NULL, all other values indicate NOT NULL.

NONULL specifies that the values in the input file are entered into the columns exactly as read (i.e. NULL values can not be entered).

DUPLICATE (default) specifies that the number of duplicates found during the load operation will be reported. NODUPLICATES means that number of duplicates will not be reported or logged.

The number of rows not loaded because of a conversion error will be reported (and logged if LOGFILE has been specified).

LOGFILE specifies a sequential file, where duplicate rows and rows not loaded because of a conversion error may be logged.

If column-specifications are given, only values for the columns which are given will be read from the input file. For each column, the sequential position for the start and the end byte of the value to assign should be specified in POS(s:e).

If a delimiter character is specified, the values for the columns which are read from the input file are expected to be delimited by the specified character.

If neither column-specifications nor a delimiter character are specified, default values for positions to read from are determined from the table definition. All columns will be given values.

The LOAD command may not be used if a transaction is active. For further information on transactions, see Chapter 6.

Examples:

```
LOAD FROM 'rooms.dat' INTO rooms NULL,DUPLICATES
LOGFILE 'rooms.dup';

LOAD FROM 'rooms2' INTO rooms NONULL (roomno POS(1:5),
roomtype POS(8:18));

LOAD FROM 'rooms.txt' INTO rooms NONULL DELIMITER ',';
```

LOG

Logs input, output or both to a specified sequential file.

Syntax

LOG INPUT|OUTPUT| INPUT,OUTPUT ON|APPEND 'filename';

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 stopped with the SET LOG OFF command and is resumed with the SET LOG ON command.

READ INPUT

Reads commands from a sequential file.

Syntax

READ [COMMAND|ALL] [INPUT FROM] 'filename';

Description

Commands and SQL statements are read from the specified file.

When READ COMMAND INPUT is specified, commands are read from the file while prompt answers are taken from the terminal (batch job, command procedure).

When READ ALL INPUT or READ INPUT is specified, both commands and prompt answers are read from the sequential file.

SET ECHO

Controls whether or not lines read during READ INPUT are echoed.

Syntax

SET ECHO ON|OFF;

Description

When echo is set to ON, lines read during READ INPUT are echoed to the terminal or batch 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 LINECOUNT

Sets the length of the terminal page.

Syntax

SET LINECOUNT|LC value;

Description

The LINECOUNT value defines the length of the terminal page.

If LINECOUNT has a value greater than zero, terminal 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 linecount 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.

Default

If BSQL is run from a batch job, LINECOUNT is zero by default. For interactive operation, the default value is machine- and terminal-dependent.

SET LINESPACE

Sets the number of blank lines between each output record.

Syntax

SET LINESPACE|LS value;

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

SET LINEWIDTH|LW value;

Description

The LINEWIDTH value defines the maximum line width for output to the terminal or batch log file.

The value for LINEWIDTH cannot be set to a value less than 20.

SET LOG

Stops or resumes logging input, output or both.

Syntax

SET [INPUT|OUTPUT|INPUT, OUTPUT] LOG OFF|ON;

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 MESSAGE

Specifies whether or not messages should be displayed.

Syntax

SET MESSAGE|MSG ON|OFF;

Description

Specifies whether or not result messages such as “One row found” etc. are written to the terminal screen or batch log file.

The default setting is ON.

SET OUTPUT

Specifies whether or not output should be displayed.

Syntax

SET OUTPUT ON|OFF;

Description

When OUTPUT is set to ON, the output from BSQL is written to the terminal or batch log file. 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

SET PAGELENGTH|PL value;

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.

SET PAGEWIDTH

Specifies the page width of the output log file.

Syntax

SET PAGEWIDTH|PW value;

Description

The PAGEWIDTH value defines the page width of the output file. The value should be >= 20. The default value is machine-dependent.

SHOW SETTINGS

Displays the current values of all set options.

Syntax

SHOW SETTINGS;

Description

Display the current values for all set options, i.e. ECHO, LINECOUNT, LINESPACE, LINEWIDTH, LOG, MESSAGE, OUTPUT, PAGELENGTH, PAGEWIDTH, TRANSACTION START, TRANSACTION ISOLATION LEVEL and TRANSACTION MODE (read only or read write).

Current server and connection names are also displayed.

UNLOAD

The UNLOAD command can be used to unload data from a table into a sequential file.

Syntax

UNLOAD TO 'file-name' FROM table-name <NULL|NONULL>
< (col-name POS(s:e), ..., col-name POS(s:e) | DELIMITER 'character' >;

Description

NULL (default) specifies that the first byte for each column value in the output file is used to indicate whether the value is NULL or not. This byte is assigned an ampersand (&) if the column from which the field is derived contains NULL, the rest of the field is filled with periods (...). Otherwise the byte is blank.

NONULL specifies that the first byte for each column value in the output file is the first data byte of the value.

If column-specifications are given, the output file will only hold values for the columns which are given. For each column the sequential position of the start and the end byte of the column value should be specified in POS(s:e). Overlapping is not controlled.

If a delimiter character is given, column data will be written to the output file delimited by the specified character. All columns will be included.

If neither column-specifications nor a delimiter character are given, default values for positions are determined by the table definition. All columns will be included.

The UNLOAD command may not be used if a transaction is active. For further information on transactions, see Chapter 6.

Example:

```
UNLOAD TO 'rooms.dat' FROM rooms;

UNLOAD TO 'rooms2' FROM rooms NONULL
  (roomno POS(1:5), roomtype POS(8:18));

UNLOAD TO 'rooms.txt' FROM rooms NONULL DELIMITER ',';
```

WHENEVER

Determines which actions should be taken in the event of an error or warning.

Syntax

WHENEVER ERROR|WARNING action<,action>;

Description

If an error or warning should occur in a file being run in batch, there are several “action” options that may be chosen to determine what should happen.

The actions can be broken down into two groups:

Execution flow

EXIT	Leaves BSQL in batch 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.

Transaction control

ROLLBACK	Abandons the transaction; no changes are made to the database.
COMMIT	Requests that the operations are executed against the database, and the changes in the database are made permanent.

The transaction control action can only be used if the execution flow is specified as EXIT. If execution flow is CONTINUE any ongoing transaction will not be affected by an error.

Default

The default value for warning is CONTINUE.

The default values for errors are EXIT, ROLLBACK in batch mode or file input mode and CONTINUE in interactive mode.

10 VARIABLES IN BSQL

Host variables are used in embedded SQL statements to pass values between the database and an application program (see the Mimer SQL Programmer's Manual). Host variables are also supported in BSQL, to facilitate interactive design and testing of SQL statements intended for use in embedded SQL application programs. In BSQL, the host variables serve as parameter markers, and the user is prompted for parameter values when the statement is executed.

Host variables may be 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), and 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.

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.

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.

10.1 Host variables

When host variables are used in BSQL, BSQL prompts for the variable values, for example:

```
SQL>SELECT * FROM HOTEL WHERE CITY = :CITY;
CITY: STOCKHOLM
```

This statement is then executed as

```
SQL>SELECT * FROM HOTEL WHERE CITY = 'STOCKHOLM';
```

<u>Note:</u> The entered variable is **not** enclosed between apostrophes, in contrast to the corresponding string value. Variables enclosed in apostrophes will be interpreted as literal strings.

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 BOOK_GUEST SET ARRIVE    = :ARRIVE:NULL,
SQL>                      DEPART    = :DEPART:NULL
SQL>  WHERE RESERVATION = 1348;
Null:N
ARRIVE: 2001-04-23
Null:Y
```

<u>Note:</u> The prompts appear in the order in which the variables are used in the statement. In the example above, the ARRIVE value will be updated to 2001-04-23 and the DEPART value will be set to NULL.

11 ERROR HANDLING

11.1 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.

11.1.1 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 quotation marks instead of apostrophes around the string constant:

```
SELECT  "Hotel:",NAME
FROM    HOTEL;
```

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.

11.1.2 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.

Some examples of errors and resulting error messages are listed below.

```
SELECT  AVG(NAME) FROM HOTEL;
```

Error message:

```
SELECT  AVG(NAME) FROM HOTEL;
            ^
Invalid operand type, expected type is NUMERIC or INTERVAL
```

```
SELECT  NAME FROM HOTEL
WHERE   CITY ON ('STOCKHOLM','UPPSALA');
```

Error message:

```
SELECT  NAME FROM HOTEL
WHERE   CITY ON ('STOCKHOLM','UPPSALA');
             ^
Syntax error, 'ON' assumed to mean 'IN'
```

In the following example, the error analysis is misleading:

```
SELECT  NAME FROM HOTEL
WJERE   HOTELCODE = 'LAP';
```

Error message:

```
SELECT  NAME FROM HOTEL
WJERE   HOTELCODE = 'LAP';
        ^
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.

11.2 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:

-1500	Illegal value for <%>
-1400	Invalid numerical argument
-1300	Only select statements can be used with PRINT
-1200	Previous perform file is not finished
-1101	Disk space exhausted
-1009	Unspecified file open error
-1008	** Installation dependent **
-1007	** Installation dependent **
-1006	Disk space exhausted
-1005	Maximum number of opened files exceeded
-1004	File locked
-1003	File protection violation
-1002	File not found
-1001	Syntax error in file name
-999	Too long statement
-900	No buffer saved
-801	Pending transaction, Commit or Rollback
-800	Load/unload is not allowed within a transaction
-777	Maximum header length exceeded
-776	Maximum record length <%> exceeded
-701	Help topic not found
-700	Help databank not installed or inaccessible
-666	Space area exhausted
-600	The number of host variables cannot exceed 20
-400	Record too large for one page (<%> lines required) Increase value of LC/PL or set them to zero
-300	Failed to read dictionary
-207	Too many parameters
-206	Unexpected end of command
-205	Invalid numerical literal
-204	Filenames must be enclosed in apostrophes
-203	String expected
-202	Undefined keyword
-201	Syntax error
-104	Missing statement terminator (@)
-103	Missing semicolon
-102	<%> command not valid in this context
-101	Ambiguous command <%>
-100	Undefined command <%>

-5	Conflict. One of COMMIT or ROLLBACK and EXIT or CONTINUE
-3	Too many files have been opened
-2	File could not be opened
-1	String exceeds 256 characters which is not allowed

A EXAMPLE DATABASE

A simple example database is used throughout this manual to illustrate the use of Mimer SQL. It is based upon an imaginary company that owns a chain of hotels.

The database is created in the databank HOTELDB.

The schema for the example database is created by the ident HOTELADM.

This example database is provided with the Mimer SQL installation so that you may try out the examples yourself (if you do not have the example database, ask your Mimer SQL system administrator to generate it). The tables shown here provide an easy reference for the examples in the manual. The statements used to create this database are also shown in this appendix.

A.1 Tables in the example database

Tables in the example database are described in this section.

The table descriptions are set up as follows:

- The first column lists the table name and the column names.
- The second column shows which columns which make up the primary key (*).
- The third column shows the columns that are foreign keys (*f*). Refer to the CREATE statements later in this section for a full definition of foreign keys in the database.
- The fourth column shows the column data type. CHAR(n) is a character string of length *n* bytes. INT(p) specifies an integer of up to *p* digits long. DEC(p,s) specifies numbers of up to *p* digits long, of which *s* follow the decimal point. DATE is a date in the Gregorian calendar in the form YYYY-MM-DD. TIME(s) is a time on an unspecified day, in the form HH:MM:SS, with *s* digits following the decimal point in the seconds value.
- The fifth column explains the column contents.

A.2 Table descriptions

HOTEL				
HOTELCODE	*		CHAR(4)	Hotel identity code
NAME			CHAR(15)	Hotel name
CITY			CHAR(15)	Location

ROOMSTATUS				
STATUS	*		CHAR(10)	Room status

ROOMTYPES				
ROOMTYPE	*		CHAR(6)	Room type
DESCRIPTION			VARCHAR(40)	Room description

ROOMS				
ROOMNO	*		CHAR(7)	Room number
HOTELCODE		*f*	CHAR(4)	Hotel identity code
ROOMTYPE		*f*	CHAR(6)	Room type
STATUS		*f*	CHAR(10)	Room status

ROOM_PRICES				
HOTELCODE	*	*f*	CHAR(4)	Hotel identity code
ROOMTYPE	*	*f*	CHAR(6)	Room type
FROM_DATE	*		DATE	Date when price becomes valid
TO_DATE			DATE	Date until which price is valid
PRICE			INT(4)	Cost of room per day

CHARGES				
CHARGE_CODE	*		CHAR(3)	Charge code
DESCRIPTION			CHAR(25)	Cost description
CHARGE_PRICE			INT(4)	Price charged for room

BOOK_GUEST				
RESERVATION	*		INT(5)	Guest reference number
BOOKING_DATE			DATE	Date of booking
HOTELCODE		*f*	CHAR(4)	Hotel identity code
ROOMTYPE		*f*	CHAR(6)	Room type
COMPANY			VARCHAR(100)	Name of company reserving room
TELEPHONE			CHAR(15)	Telephone number of above
RESERVED_FNAME			CHAR(25)	First name of expected guest
RESERVED_LNAME			CHAR(25)	Last name of expected guest
ARRIVE			DATE	Expected check-in date
DEPART			DATE	Expected check-out date
GUEST_FNAME			CHAR(25)	Guest first name
GUEST_LNAME			CHAR(25)	Guest last name
ADDRESS			VARCHAR(50)	Guest address
CHECKIN			DATE	Actual check-in date
CHECKOUT			DATE	Actual check-out date
ROOMNO		*f*	CHAR(7)	Room number
PAYMENT			CHAR(10)	Payment type

BILL				
RESERVATION		*f*	INT(5)	Guest reference number
ON_DATE			TIMESTAMP(0)	Billing date and time
CHARGE_CODE		*f*	CHAR(3)	Charge code
COST			INT(4)	Cost of stay

WAKE_UP				
ROOMNO	*	*f*	CHAR(7)	Room number
WAKE_DATE	*		DATE	Wake up date
WAKE_TIME			TIME	Wake up time

EXCHANGE_RATE				
CURRENCY	*		CHAR(3)	Currency
RATE			DEC(6,3)	Exchange rate

A.3 The tables

This section illustrates the contents of the tables in the example database. Only partial data is shown for some tables.

HOTEL		
HOTELCODE	NAME	CITY
LAP	LAPONIA	STOCKHOLM
SKY	SKYLINE	UPPSALA
STG	ST. GEORGE	STOCKHOLM
WIN	Winston	London
WIND	WINSTON	COPENHAGEN
WINS	WINSTON	GOTHENBURG

ROOMSTATUS
STATUS
UNKNOWN
FREE
KEY OUT
MAINT

ROOMTYPES	
ROOMTYPE	DESCRIPTION
NSDBLB	NO SMOKING - DOUBLE WITH BATH
NSDBLS	NO SMOKING - DOUBLE WITH SHOWER
NSSGLB	NO SMOKING - SINGLE WITH BATH
NSSGLS	NO SMOKING - SINGLE WITH SHOWER
SDBLB	SMOKING - DOUBLE WITH BATH
SDBLS	SMOKING - DOUBLE WITH SHOWER
SSGLB	SMOKING - SINGLE WITH BATH
SSGLS	SMOKING - SINGLE WITH SHOWER

ROOMS			
ROOMNO	HOTELCODE	ROOMTYPE	STATUS
LAP110	LAP	SSGLS	FREE
LAP211	LAP	NSDBLB	UNKNOWN
LAP309	LAP	NSSGLS	UNKNOWN
...	...	...	
SKY117	SKY	NSSGLS	UNKNOWN
SKY121	SKY	NSDBLS	MAINT
...	...	...	
SKY111	SKY	SSGLB	KEY OUT
SKY114	SKY	SSGLB	UNKNOWN
...	...	...	
WIND308	WIND	NSSGLB	UNKNOWN
WIND524	WIND	SDBLB	UNKNOWN
...	...	...	
WINS108	WINS	NSDBLB	FREE
WINS109	WINS	NSSGLB	UNKNOWN
WINS116	WINS	NSDBLB	UNKNOWN

ROOM_PRICES				
HOTELCODE	ROOMTYPE	FROM_DATE	FROM_DATE	PRICE
LAP	NSSGLS	1997-11-15	1998-03-10	640
LAP	NSSGLS	1997-08-08	1997-11-14	680
...	...	...	...	...
SKY	NSSGLS	1997-08-08	1997-11-14	750
...	...	...	...	...
STG	NSSGLS	1997-11-15	1998-03-10	640
STG	NSSGLS	1997-08-08	1997-11-14	680

CHARGES		
CHARGE_CODE	DESCRIPTION	CHARGE_PRICE
100	LODGING	100
120	TELEPHONE	40
170	CAR PARK	70
200	RESTAURANT	250
210	MINIBAR	70
230	BAR	200
270	ROOM SERVICE	95
330	LAUNDRY	120
720	EXTRA BED	370
700	ROOM	-
900	MISCELLANEOUS	30

BOOK_GUEST				
RESERVATION	BOOKING_DATE	HOTELCODE	ROOMTYPE	COMPANY
1348	1997-06-10	LAP	NSSGLB	MIMER IT AB
1349	1997-06-10	LAP	NSSGLS	MIMER AB
1350	1997-06-11	SKY	SDBLB	SALLY WEBERT
1351	1997-06-11	SKY	NSDBLB	SALLY WEBERT
1352	1997-06-11	WINS	NSDBLB	MARK FRANCIS
1353	1997-06-11	SKY	NSSGLB	ASATRON AB
...	...	...	...	...
...	...	...	...	...

TELEPHONE	RESERVED_FNAME	RESERVED_LNAME	ARRIVE	DEPART
018-185210	STEN	JOHANSEN	1997-08-20	1997-08-22
018-185210	MATS	LINDBLOM	1997-06-30	1997-07-01
0760-57609	SALLY	WEBERT	1997-08-21	1997-08-24
0760-57609	JOHN	ALBERTSON	1997-06-11	1997-06-15
08-320668	MARK	FRANCIS	1997-06-19	1997-06-20
08-135709	BASIL	FAWCETT	1997-08-20	1997-08-22
...	...	...	...	...
...	...	...	...	...

GUEST_FNAME	GUEST_LNAME	ADDRESS
STEN	JOHANSEN	MIMERGATAN 4, UPPSALA
STEFAN	HANSEN	IDUNGATAN 24, UPPSALA
SALLY	WEBERT	KRONPARKEN 44, JOKKMOKK
ANNA	ALBERTSON	32 SPRING DRIVE, DENVER, USA
MARK	FRANCIS	VIMPELGATAN 7, SKARA
ALFRED	FIMPLEY	23 BACK NELLY VIEW, ACKWORTH
...	...	...
...	...	...

CHECKIN	CHECKOUT	ROOMNO	PAYMENT
1997-08-20	1997-08-22	STG009	EUROCARD
1997-06-30	1997-07-01	LAP206	EUROCARD
1997-08-21	1997-08-22	SKY212	CASH
1997-06-11	1997-06-15	SKY125	AM.EXPR
1997-06-19	1997-06-20	WINS103	EUROCARD
1997-08-20	1997-08-22	SKY110	CASH
...	...	...	...
...	...	...	...

BILL			
RESERVATION	ON_DATE	CHARGE_CODE	COST
1347	1997-08-21 13:38:19	100	100
1347	1997-08-21 13:38:19	120	40
...	...	...	...
1347	1997-08-21 13:38:19	120	40
...	...	...	...
1348	1997-08-21 13:38:19	230	200
...	...	...	...
1349	1997-06-30 13:38:19	170	70
1349	1997-06-30 13:38:19	900	30
...	...	...	...
1350	1997-08-21 13:38:19	100	100
...	...	...	...
1350	1997-08-21 13:38:19	230	200
1350	1997-08-21 13:38:19	330	120
1350	1997-08-21 13:38:19	100	100
1350	1997-08-21 13:38:19	120	40
1350	1997-08-21 13:38:19	270	95
...	...	...	...

WAKE_UP		
ROOMNO	WAKE_DATE	WAKE_TIME
LAP112	1997-08-22	06:00:00
LAP112	1997-08-23	07:00:00
LAP201	1997-08-23	06:45:00
LAP205	1997-08-22	08:00:00
SKY101	1997-08-22	09:00:00
SKY110	1997-08-22	07:30:00
SKY111	1997-08-22	06:00:00
SKY124	1997-08-22	06:15:00
SKY124	1997-08-23	06:15:00
SKY124	1997-08-24	06:15:00
SKY201	1997-08-22	10:00:00
SKY212	1997-08-22	04:30:00
STG009	1997-08-22	06:00:00
STG117	1997-08-22	07:00:00
STG142	1997-08-22	08:30:00
WIND401	1997-08-23	06:00:00
WIND402	1997-08-22	06:20:00
WIND514	1997-08-22	07:00:00
WINS119	1997-08-22	08:00:00
WINS120	1997-08-22	07:30:00
WINS121	1997-08-22	06:20:00

EXCHANGE_RATE	
CURRENCY	RATE
DEM	0.2230
DKK	0.8495
FIM	0.6560
FRF	0.7420
GBP	0.0810
ITL	206.82
JPY	16.38
NOK	0.8815
SEK	1.000
USD	0.1330

A.4 CREATE statements for example database

The following statements were used to create the tables in the example database. Only the CREATE statements are listed here.

```
CREATE DATABANK HOTELDB
       OF 60 PAGES
       IN 'HOTELDB'
       WITH TRANS OPTION;


CREATE DOMAIN HOTELCODE
       AS CHARACTER(4);


CREATE DOMAIN STATUS
       AS CHARACTER(10)
       DEFAULT 'UNKNOWN';


CREATE DOMAIN ROOMTYPE
       AS CHARACTER(6)
       DEFAULT '-ND-';


CREATE DOMAIN ROOMNO
       AS CHARACTER(7);


CREATE DOMAIN PERSONNAME
       AS CHARACTER(25);


CREATE DOMAIN NUMBER
       AS INTEGER(3)
       DEFAULT 0;


CREATE DOMAIN BOOK_RATE
       AS DECIMAL(3,2)
       DEFAULT 1.10;


CREATE TABLE HOTEL (HOTELCODE   HOTELCODE  NOT NULL,
                    NAME        CHAR(15)   NOT NULL,
                    CITY        CHAR(15)   NOT NULL,
       PRIMARY KEY (HOTELCODE))
       IN HOTELDB;


CREATE TABLE ROOMSTATUS (STATUS STATUS NOT NULL,
       PRIMARY KEY (STATUS)) IN HOTELDB;


CREATE TABLE ROOMTYPES (ROOMTYPE     ROOMTYPE     NOT NULL,
                        DESCRIPTION  VARCHAR(40)  NOT NULL,
       PRIMARY KEY (ROOMTYPE))
       IN HOTELDB;


CREATE TABLE ROOMS (ROOMNO     ROOMNO     NOT NULL,
                    HOTELCODE  HOTELCODE  NOT NULL,
                    ROOMTYPE   ROOMTYPE   NOT NULL,
                    STATUS     STATUS     NOT NULL,
       PRIMARY KEY (ROOMNO),
       FOREIGN KEY (HOTELCODE) REFERENCES HOTEL,
       FOREIGN KEY (ROOMTYPE)  REFERENCES ROOMTYPES,
       FOREIGN KEY (STATUS)    REFERENCES ROOMSTATUS)
       IN HOTELDB;
```

```
CREATE TABLE ROOM_PRICES (HOTELCODE  HOTELCODE  NOT NULL,
                          ROOMTYPE   ROOMTYPE   NOT NULL,
                          FROM_DATE  DATE       NOT NULL,
                          TO_DATE    DATE       NOT NULL,
                          PRICE      INTEGER(4),
       PRIMARY KEY (HOTELCODE,ROOMTYPE,FROM_DATE),
       FOREIGN KEY (HOTELCODE) REFERENCES HOTEL,
       FOREIGN KEY (ROOMTYPE)  REFERENCES ROOMTYPES)
       IN HOTELDB;


CREATE TABLE CHARGES (CHARGE_CODE   CHAR(3)   NOT NULL,
                      DESCRIPTION   CHAR(25)  NOT NULL,
                      CHARGE_PRICE  INTEGER(4),
       PRIMARY KEY (CHARGE_CODE))
       IN HOTELDB;


CREATE TABLE BOOK_GUEST (RESERVATION     INTEGER(5)    NOT NULL,
                         BOOKING_DATE    DATE
                           DEFAULT  CURRENT_DATE       NOT NULL,
                         HOTELCODE       HOTELCODE     NOT NULL,
                         ROOMTYPE        ROOMTYPE      NOT NULL,
                         COMPANY         VARCHAR(100)  NOT NULL,
                         TELEPHONE       CHAR(15),
                         RESERVED_FNAME  PERSONNAME,
                         RESERVED_LNAME  PERSONNAME,
                         ARRIVE          DATE          NOT NULL,
                         DEPART          DATE          NOT NULL,
                         GUEST_FNAME     PERSONNAME,
                         GUEST_LNAME     PERSONNAME,
                         ADDRESS         VARCHAR(50),
                         CHECKIN         DATE,
                         CHECKOUT        DATE,
                         ROOMNO          ROOMNO,
                         PAYMENT         CHAR(10),
       PRIMARY KEY (RESERVATION),
       FOREIGN KEY (HOTELCODE) REFERENCES HOTEL,
       FOREIGN KEY (ROOMTYPE)  REFERENCES ROOMTYPES,
       FOREIGN KEY (ROOMNO)    REFERENCES ROOMS,
       CHECK (ARRIVE < DEPART AND CHECKIN <= CHECKOUT))
       IN HOTELDB;


CREATE TABLE BILL (RESERVATION  INTEGER(5)   NOT NULL,
                   ON_DATE      TIMESTAMP(0) NOT NULL,
                   CHARGE_CODE  CHAR(3)      NOT NULL,
                   COST         INTEGER(4)
                     DEFAULT  NULL,
       FOREIGN KEY (RESERVATION) REFERENCES BOOK_GUEST,
       FOREIGN KEY (CHARGE_CODE) REFERENCES CHARGES)
       IN HOTELDB;


CREATE TABLE WAKE_UP(ROOMNO     ROOMNO  NOT NULL,
                     WAKE_DATE  DATE    NOT NULL,
                     WAKE_TIME  TIME    NOT NULL,
       PRIMARY KEY (ROOMNO,WAKE_DATE),
       FOREIGN KEY (ROOMNO) REFERENCES ROOMS)
       IN HOTELDB;


CREATE TABLE EXCHANGE_RATE (CURRENCY  CHAR(3)      NOT NULL,
                            RATE      DECIMAL(6,3),
       PRIMARY KEY (CURRENCY))
       IN HOTELDB;
```

```
--
-- PROCEDURE TO ENTER THE CHARGE FOR LODGING ON A GUEST'S BILL
--
@
CREATE PROCEDURE ADD_LODGING (IN IN_RESERVATION INTEGER)
MODIFIES SQL DATA
BEGIN
   DECLARE P_PRICE, P_DAYS INTEGER;
   DECLARE P_CHECKIN DATE;
--
-- FIND PRICE OF ROOM
--
   SELECT PRICE INTO P_PRICE
     FROM ROOM_PRICES, BOOK_GUEST
    WHERE BOOK_GUEST.RESERVATION = IN_RESERVATION
      AND ROOM_PRICES.ROOMTYPE = BOOK_GUEST.ROOMTYPE
      AND ROOM_PRICES.HOTELCODE = BOOK_GUEST.HOTELCODE
      AND FROM_DATE <= CURRENT_DATE
      AND TO_DATE >= CURRENT_DATE;
--
-- FIND LENGTH OF STAY
--
   SELECT CAST((CHECKOUT-CHECKIN) DAY AS INTEGER), CHECKIN
     INTO P_DAYS, P_CHECKIN
     FROM BOOK_GUEST WHERE RESERVATION=IN_RESERVATION;

   BEGIN
      DECLARE P_COUNTER INTEGER DEFAULT 0;
      WHILE P_COUNTER < P_DAYS DO
         INSERT INTO BILL VALUES
               (IN_RESERVATION,
                CAST(P_CHECKIN+CAST(P_COUNTER AS INTERVAL DAY)
                  AS TIMESTAMP),
                '100',
                P_PRICE);
         SET P_COUNTER = P_COUNTER+1;
      END WHILE;
   END;
END
@



--
-- PROCEDURE TO LIST ALL ROOMS THAT HAVE REQUIRED A WAKE-UP
-- CALL WITHIN THE GIVEN INTERVAL
--
@
CREATE PROCEDURE WAKE_UP(IN WAKE_UP INTERVAL MINUTE(4)) VALUES(CHAR(7))
READS SQL DATA
BEGIN
   DECLARE WAKE CURSOR FOR SELECT ROOMNO
                           FROM WAKE_UP
                           WHERE
                             CAST(CAST(WAKE_DATE AS CHAR(10)) || ' '
                             || CAST(WAKE_TIME AS CHAR(10)) AS TIMESTAMP)
                           BETWEEN LOCAL_TIMESTAMP
                               AND LOCAL_TIMESTAMP + WAKE_UP;
   DECLARE ROOM CHAR(7);
   OPEN WAKE;
   BEGIN
      DECLARE EXIT HANDLER FOR NOT FOUND BEGIN END;
      LOOP
         FETCH WAKE INTO ROOM;
         RETURN ROOM;
      END LOOP;
   END;
   CLOSE WAKE;
END
@
```

```
--
-- PROCEDURE TO ALLOCATE A ROOM FOR A GUEST
--
@
CREATE PROCEDURE ALLOCATE_ROOM (IN IN_RESERVATION INTEGER,INOUT
OUT_ROOMNO CHAR(6))
MODIFIES SQL DATA
BEGIN
   SELECT MAX(ROOMS.ROOMNO)
     INTO OUT_ROOMNO
     FROM ROOMS,BOOK_GUEST
    WHERE BOOK_GUEST.RESERVATION = IN_RESERVATION
      AND ROOMS.HOTELCODE = BOOK_GUEST.HOTELCODE
      AND ROOMS.ROOMTYPE  = BOOK_GUEST.ROOMTYPE
      AND ROOMS.STATUS = 'FREE';

   UPDATE ROOMS
      SET STATUS = 'UNKNOWN'
    WHERE ROOMNO = OUT_ROOMNO;

   UPDATE BOOK_GUEST
      SET ROOMNO = OUT_ROOMNO
    WHERE RESERVATION = IN_RESERVATION;
END
@


--
-- PROCEDURE TO BE CALLED WHENEVER A GUEST CONSUMES ANYTHING
-- AND CHARGES IT TO HIS/HER ROOM
--
@
CREATE PROCEDURE CHARGE_ROOM(IN IN_ROOMNO CHAR(6),
                             IN IN_CHARGE_CODE CHAR(3))
MODIFIES SQL DATA
BEGIN
   DECLARE P_RESERVATION, P_PRICE, P_RC  INTEGER;

   SELECT RESERVATION
     INTO P_RESERVATION
     FROM BOOK_GUEST
    WHERE ROOMNO = IN_ROOMNO;

   GET DIAGNOSTICS P_RC = ROW_COUNT;
   IF P_RC = 0 THEN
      SIGNAL SQLSTATE '05001';
   END IF;

   SELECT CHARGE_PRICE
     INTO P_PRICE
     FROM CHARGES
    WHERE CHARGE_CODE = IN_CHARGE_CODE;

    GET DIAGNOSTICS P_RC = ROW_COUNT;
    IF P_RC = 0 THEN
       SIGNAL SQLSTATE '05002';
    END IF;

    BEGIN
       DECLARE EXIT HANDLER FOR SQLEXCEPTION
       BEGIN
          SIGNAL SQLSTATE '05003';
       END;
       INSERT INTO BILL VALUES
                (P_RESERVATION,
                 LOCAL_TIMESTAMP,
                 IN_CHARGE_CODE,
                 P_PRICE);
    END;
END
@
```

```
--
-- PROCEDURE TO FREE UP A ROOM
--
@
CREATE PROCEDURE DEALLOC_ROOM (IN IN_RESERVATION INTEGER)
MODIFIES SQL DATA
BEGIN
   DECLARE P_ROOMNO CHAR(7);

   SELECT ROOMNO
     INTO P_ROOMNO
     FROM BOOK_GUEST
    WHERE RESERVATION = IN_RESERVATION;

   UPDATE ROOMS
      SET STATUS = 'FREE'
    WHERE ROOMNO = P_ROOMNO;

   UPDATE BOOK_GUEST
      SET ROOMNO = NULL
    WHERE RESERVATION = IN_RESERVATION;
END
@


--
-- PROCEDURE TO FIND FREE ROOMS FOR A RESERVATION REQUEST
--
@
CREATE PROCEDURE FREEQ (IN IN_HOTELCODE CHAR(3),
                        IN IN_ROOMTYPE CHAR(6),
                        IN IN_ARRIVE DATE,
                        IN IN_DEPART DATE,
                        OUT OUT_ROOMS INTEGER)
READS SQL DATA
BEGIN
   DECLARE P_RESERVED,P_AVAIL INTEGER;

   SELECT COUNT(RESERVATION)
     INTO P_RESERVED
     FROM BOOK_GUEST
    WHERE ARRIVE <= IN_ARRIVE
      AND DEPART >= IN_DEPART
      AND ROOMTYPE = IN_ROOMTYPE
      AND HOTELCODE = IN_HOTELCODE;

   SELECT COUNT(ROOMNO)
     INTO P_AVAIL
     FROM ROOMS
    WHERE ROOMTYPE = IN_ROOMTYPE
      AND HOTELCODE = IN_HOTELCODE;

   SET OUT_ROOMS = P_AVAIL - P_RESERVED;
END
@


--
-- PROCEDURE TO PROCESS A GUEST CHECKING OUT
--
@
CREATE PROCEDURE GUEST_LEAVES(IN IN_RESERVATION INTEGER)
MODIFIES SQL DATA
BEGIN
   CALL ADD_LODGING(IN_RESERVATION);
   CALL DEALLOC_ROOM(IN_RESERVATION);
END
@
```

```
--
-- AT THE DESK OF THE HOTEL THE STAFF USE A VIEW "FREE_ROOMS" TO FIND
-- FREE ROOMS, AS IT IS A JOINVIEW IT IS NOT UPDATABLE.
--

CREATE VIEW FREE_ROOMS AS SELECT R.ROOMNO,R.HOTELCODE,T.DESCRIPTION FROM
   ROOMS R,ROOMTYPES T
   WHERE R.ROOMTYPE=T.ROOMTYPE
      AND R.STATUS='FREE';

@
CREATE TRIGGER FREEUPDATE INSTEAD OF UPDATE ON FREE_ROOMS
REFERENCING NEW TABLE AS N
BEGIN ATOMIC
  .UPDATE ROOMS
      SET STATUS = 'USED'
   WHERE ROOMS.ROOMNO =(SELECT ROOMNO FROM N);
END
@



--
-- THE STATUS OF A ROOM IS KEPT IN THE ROOMS TABLE, NOW THE HOTEL
-- POLICY IS THAT YOU MAY NEVER DO ANY MAINTAINANCE ON A ROOM WHEN THE
-- KEY IS OUT
--
-- THIS TRIGGER PREVENTS SETTING THE STATUS 'MAINT' WHEN IT IS CURRENTLY
-- 'KEY OUT'
--

@
CREATE TRIGGER SETMAINT  AFTER  UPDATE ON ROOMS
REFERENCING NEW TABLE AS N
   OLD TABLE AS O
BEGIN ATOMIC
   IF  EXISTS (SELECT STATUS FROM O WHERE STATUS='KEY OUT')
   AND EXISTS (SELECT STATUS FROM N WHERE STATUS='MAINT')  THEN
      SIGNAL SQLSTATE  VALUE '07020';
   END IF ;
END
@



--
-- THIS TRIGGER ONLY WORKS IF YOU UPDATE ONLY ONE ROOM AT A TIME, TO GET
-- IT WORKING FOR MULTI-ROW-UPDATES YOU WOULD HAVE TO DECLARE 2 CURSORS
-- AND STEP IN PARALLEL OVER THE O AND N TABLE COMPARING VALUES.
--
-- IN THE BILL TABLE THERE MAY NEVER BE MORE THAN ONE
-- CHARGE FOR EACH DAY FOR THE CHARGES
-- THIS TRIGGER PREVENTS SUCH INSERTS
--
@
CREATE TRIGGER BILLINSERT AFTER INSERT ON BILL
REFERENCING NEW TABLE AS N
BEGIN ATOMIC
   IF EXISTS (SELECT *
              FROM BILL,N
              WHERE BILL.RESERVATION = N.RESERVATION
                AND BILL.ON_DATE = N.ON_DATE
                AND BILL.CHARGE_CODE = N.CHARGE_CODE
                AND N.CHARGE_CODE IN ('100','170','720') )  THEN
      SIGNAL SQLSTATE  VALUE '07020';
   END IF;
END
@
```

```
--
-- WHEN A CUSTOMER PAYS THE BILL RECORDS IN THE BILL TABLE ARE
-- DELETED.  IF THE CUSTOMER PAYS FOR LODGING (CODE=100) MAKE SURE THE
-- ROOM GETS THE STATUS 'FREE'
--

@
CREATE TRIGGER BILLDELETE AFTER DELETE ON BILL
REFERENCING OLD TABLE AS OLDROWS
BEGIN ATOMIC
    UPDATE ROOMS
       SET STATUS='FREE'
          WHERE ROOMS.ROOMNO = (SELECT BOOK_GUEST.ROOMNO
                                FROM BOOK_GUEST, OLDROWS
                                WHERE OLDROWS.RESERVATION
                                    = BOOK_GUEST.RESERVATION
                                   AND OLDROWS.CHARGE_CODE='100');
END
@


--
-- HOTEL MANAGEMENT DECIDES THAT A COLUMN "RATING" SHOULD BE ADDED TO THE
-- HOTEL INFORMATION. A NEW TABLE HOTELN IS DEFINED THAT CONTAINS THIS
-- NEW COLUMN.ALL NEW APPLICATIONS SHOULD USE THIS TABLE.
--

CREATE TABLE HOTELN(
   HOTELCODE HOTELCODE NOT NULL,
   NAME CHAR(15) NOT NULL,
   CITY CHAR(15) NOT NULL,
   RATING CHAR(5),
   PRIMARY KEY(HOTELCODE) )
  IN  HOTELDB;


--
-- IN ORDER TO GET ALL OLD APPLICATIONS WORKING A VIEW HOTEL IS DEFINED
--
CREATE VIEW HOTEL AS SELECT  HOTELCODE, NAME, CITY FROM HOTELN;


--
-- BY DEFINING A INSTEAD OF INSERT TRIGGER ON THE VIEW,
-- WE CAN GET THE EFFECT THAT WHENEVER AN OLD APPLICATION
-- INSERTS THINGS IN THE HOTEL VIEW (THE OLD APPLICATIONS SEES IT AS A
-- TABLE) THE VALUE '-' IS INSERTED IN THE NEW HOTELN TABLE!
--

@
CREATE TRIGGER HOTINSERT INSTEAD OF INSERT ON HOTEL
REFERENCING NEW TABLE AS NEWROWS
BEGIN ATOMIC
   INSERT INTO HOTELN  SELECT HOTELCODE, NAME, CITY, '-' FROM NEWROWS ;
END
@
```

INDEX

M

N

O

P

Q

R

S

W