



MIMER/SQL

User's Manual

Version 8.1

Copyright © 1998 Sysdeco Mimer AB

MIMER/SQL version 8.1 User's Manual

June, 1998

Copyright © 1998 Sysdeco Mimer AB.

Published by Sysdeco Mimer AB,

P.O. Box 1713,

S-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 Sysdeco Mimer AB, Uppsala, Sweden.

All rights reserved under international copyright conventions.

The contents of this manual may be printed in limited quantities for use at a Mimer installation site. No parts of the manual may be reproduced for sale to a third party.

FOREWORD

Documentation objectives

This manual is intended primarily for users of MIMER Batch SQL (BSQL), and for users with little or no experience of SQL (Structured Query Language). It describes how to use SQL for creating and manipulating the database contents, and includes a detailed reference of the facilities within BSQL.

This manual does not attempt to give an exhaustive description of SQL. Refer to the *MIMER/SQL Reference Manual* for a complete syntax description of the SQL statements.

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 MIMER Batch SQL 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 database management system and 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 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 database system.

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

Chapter 8 describes how to manage access rights and privileges in the database.

Chapters 9-11 describe the MIMER Batch SQL (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 contains machine dependent information.

Appendix B lists the structure and contents of an example database provided with the MIMER distribution and used in the examples in this manual.

Related MIMER publications

- **MIMER/SQL Reference Manual** contains a complete description of the syntax and usage of all statements in MIMER/SQL. The Reference Manual 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 Shadowing Handbook** describes the optional Shadowing module, which allows several concurrent copies of a databank to be updated at the same time. This provides extra resilience to disk crashes and allows “backup on the fly”. The SQL statements which are part of the Shadowing API are described in the SQL Reference Manual.
- **MIMER System Management Handbook** describes system administration functions, including export/import, backup/restore, and the statistics functionality. The information in this manual is used primarily by the system administrator, and is not required by application developers. The SQL statements which are part of the System Management API are described in the SQL Reference Manual.
- **MIMER Guides** comprise documentation containing platform-specific administration and usage information. A set of one or more guides is provided, where required, for each platform on which MIMER is supplied.
- **MIMER Release Notes** contain platform-specific information relating to the MIMER 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 “SQL3”.

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 9075:1992(E) Information technology - Database languages - SQL. This document contains the standard referred to as SQL2.

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 database language routines.

CAE Specification, Structured Query Language (SQL). X/Open document number: C201. ISBN: 1 872630 58 8. This document contains the X/Open-92-SQL specification.

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 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	
2	BASIC CONCEPTS OF MIMER/SQL	
2.1	The MIMER relational database	2-1
2.1.1	The data dictionary	2-1
2.1.2	Databanks	2-2
2.1.3	Idents	2-2
2.1.4	Tables	2-3
2.1.5	Base tables and views	2-5
2.1.6	Primary keys and indexes	2-6
2.1.7	Procedures	2-7
2.1.8	Modules	2-7
2.1.9	Synonyms	2-8
2.1.10	Shadows.....	2-8
2.2	Data integrity.....	2-9
2.2.1	Domains.....	2-9
2.2.2	Foreign keys - referential integrity.....	2-9
2.2.3	Check conditions	2-10
2.2.4	Check options in view definitions.....	2-11
2.3	Access rights and privileges	2-11
2.4	MIMER/SQL statements.....	2-12
3	MANAGING DATABASE CONNECTIONS	
3.1	Database connections	3-1
3.1.1	Connecting to a database	3-1
3.1.2	Changing connections.....	3-2
3.1.3	Disconnecting	3-2
3.2	Program idents - ENTER and LEAVE.....	3-3
4	RETRIEVING DATA FROM TABLES	
4.1	Retrieval from single tables.....	4-1
4.1.1	Simple retrieval.....	4-1
4.1.2	Setting column labels.....	4-3
4.1.3	Eliminating duplicate values.....	4-3
4.1.4	Selecting specific rows	4-5
4.1.5	Retrieving computed values.....	4-10
4.1.6	Using set functions.....	4-11
4.1.7	Grouped set functions: the GROUP BY clause	4-13
4.1.8	Selecting groups: the HAVING clause	4-14
4.1.9	Ordering the result table	4-15
4.1.10	Using scalar functions.....	4-16
4.1.11	Using CASE expression.....	4-18
4.1.12	Using CAST specification	4-19
4.1.13	Datetime arithmetic and functions	4-19
4.2	Retrieving data from more than one table	4-21
4.2.1	The join condition.....	4-21

4.2.2	Simple joins	4-23
4.2.3	Outer joins	4-25
4.2.4	Nested selects	4-26
4.2.5	Ordering nested queries	4-28
4.2.6	Correlation names	4-29
4.2.6.1	Simplifying complex queries	4-29
4.2.7	Retrieving with EXISTS, NOT EXISTS	4-31
4.2.8	Retrieval with ALL, ANY, SOME	4-33
4.2.9	Union queries	4-34
4.3	Handling NULL values	4-37
4.3.1	Searching for NULL	4-37
4.3.2	Null values in ALL, ANY, IN and EXISTS queries	4-38
4.4	Conceptual description of the selection process	4-40
5	DATA MANIPULATION	
5.1	Inserting data	5-1
5.1.1	Inserting explicit values	5-3
5.1.2	Inserting with a subselect	5-3
5.1.3	Inserting NULL values	5-4
5.2	Updating tables	5-4
5.3	Deleting rows from tables	5-5
5.4	Calling procedures	5-5
5.5	Updatable views	5-6
6	MANAGING TRANSACTIONS	
6.1	Transactions	6-1
6.2	Logging	6-1
6.3	Handling transactions	6-2
6.3.1	Transaction handling in BSQL	6-3
6.3.2	Optimizing transactions	6-3
6.3.3	Consistency within a transaction	6-3
6.3.4	Default transaction options	6-4
7	DEFINING THE DATABASE	
7.1	Creating idents	7-1
7.2	Creating databanks	7-2
7.3	Creating domains	7-3
7.3.1	Domains with a default value	7-3
7.3.2	Domains with a check clause	7-4
7.4	Creating tables	7-4
7.4.1	Column definitions	7-6
7.4.2	The primary key	7-6
7.4.3	Alternate key	7-6
7.4.4	Foreign keys - referential integrity	7-6
7.4.5	Check conditions	7-8
7.5	Creating procedures and modules	7-9
7.6	Creating views	7-10
7.6.1	Check options	7-11
7.7	Creating secondary indexes	7-12
7.8	Creating synonyms	7-13
7.9	Commenting objects	7-14
7.10	Altering databanks, tables and idents	7-14
7.10.1	Altering a databank	7-14
7.10.2	Altering tables	7-15
7.10.3	Altering idents	7-16
7.10.4	Objects which may not be altered	7-16
7.11	Dropping objects from the database	7-17
7.11.1	Dropping databanks and tables	7-17

7.11.2	Dropping domains	7-18
7.11.3	Dropping idents	7-18
7.11.4	Dropping procedures and modules	7-19
8	DEFINING PRIVILEGES	
8.1	Ident hierarchy.....	8-2
8.2	Granting privileges	8-3
8.2.1	Granting system privileges.....	8-3
8.2.2	Granting object privileges.....	8-3
8.2.3	Granting access privileges	8-4
8.3	Revoking privileges.....	8-5
8.3.1	Revoking system privileges	8-5
8.3.2	Revoking object privileges	8-6
8.3.3	Revoking access privileges	8-6
8.3.4	Recursive effects of revoking privileges.....	8-7
9	BSQL COMMANDS	
9.1	Running BSQL	9-1
9.1.1	Running BSQL from a batch job	9-1
9.1.2	Running BSQL via the terminal.....	9-2
9.2	BSQL commands	9-3
	CLOSE	9-4
	DESCRIBE	9-4
	EXIT	9-10
	HELP.....	9-10
	LIST	9-11
	LOAD.....	9-14
	LOG	9-15
	READ INPUT	9-15
	SET ECHO.....	9-16
	SET LINECOUNT.....	9-16
	SET LINESPACE	9-17
	SET LINEWIDTH	9-17
	SET LOG	9-17
	SET MESSAGE.....	9-18
	SET OUTPUT.....	9-18
	SET PAGELength	9-18
	SET PAGEWIDTH.....	9-19
	SHOW SETTINGS	9-19
	UNLOAD	9-20
	WHENEVER	9-21
10	VARIABLES IN BSQL	
10.1	Host variables.....	10-2
11	ERROR HANDLING	
11.1	Errors in BSQL	11-1
11.1.1	Semantic errors	11-1
11.1.2	Syntax errors.....	11-2
11.2	Error messages	11-3
A	MACHINE-DEPENDENT INFORMATION	
B	EXAMPLE DATABASE	
B.1	Tables in the example database	B-1
B.2	Table descriptions	B-2
B.3	The tables	B-4
B.4	CREATE statements for example database.....	B-7

1 INTRODUCTION

MIMER is an advanced database management system developed by Sysdeco Mimer 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:

- Batch SQL (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 can support many of the tools available on the platforms supporting ODBC (e.g. Windows, Unix).

This manual describes the usage of SQL in the Batch SQL environment. Embedded SQL is described in the *MIMER/SQL Programmer's Manual*. A full description of the syntax and function of SQL statements is given in the *MIMER/SQL Reference Manual*.

2 BASIC CONCEPTS OF MIMER/SQL

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

2.1 The MIMER 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 database is "relational", which means that the information in the database is presented to the user in the form of tables. These tables represent a *logical* description of the contents of the database. The actual physical storage format may well be something else, and is of no significance to the database user.

The term database refers to the entire collection of information in a MIMER system. In addition to the information itself, this term includes the *data dictionary*, which is a set of tables describing the organization of the database, used primarily by the database management system itself. The database, although located on a single physical system, may be accessed from several distinct systems, 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 a 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 is part of a MIMER database, it contains information on all objects stored in the database and their inter-relationships to one another. The data dictionary stores information about:

- Databanks
- Access rights and privileges
- Views
- Indexes
- Shadows
- Modules
- Idents
- Tables
- Domains
- Synonyms
- Procedures

These objects can be divided into two groups:

- System objects are global to the database. System object names must be unique for each object type since they are common to all users. System objects include databanks, shadows and idents.
- Private objects are 'owned' by their creator and have names that are local to the creator. Two different users may create two different objects with the same name as long as they are not both system objects. Private objects include tables, views, domains, indexes, synonyms, procedures and modules.

Private objects are fully identified by the name of the creator and the name of the object: *creator.object*. Conflicts arising from the use of the same object name by two users are avoided in this way. In any context, a private object name without explicit reference to the creator is assumed to belong to the current user.

2.1.2 Databanks

The term databank refers to the physical file where a collection of tables is stored. A database may include any number of databanks. There are two types of databanks in the MIMER system:

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

The division of tables between different user databanks is a matter of file organization on the host computer, and does not affect the way database contents are presented to the user. Except in special situations (such as creating tables), databanks are completely invisible to the user.

Note: Backup and Restore in MIMER can be performed on a per-databank basis rather than on the entire database file base (see Chapter 6 of the *MIMER 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 idents in a MIMER database:

- **User idents** identify individual users. User idents can connect to a MIMER database, a user ident'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 authorized to use the system.

- **OS_USER idents** are idents which allow the user currently logged in to the operating system to access the MIMER database without providing a username or password. For example, if the current operating system user is ALBERT, and ALBERT is defined as an OS_USER in MIMER, ALBERT may start BSQL (or the Utilities program) and connect directly to MIMER simply by pressing <return> at the Username: prompt. If an OS_USER ident is defined with a password in MIMER, the ident may connect to MIMER in the same way as any other user ident (i.e. by providing a username and password). An OS_USER ident may not have the same name as another user ident in the database.
- **Program idents** may not connect to MIMER, but may be entered from within an application program by using the ENTER statement. The ENTER statement must be preceded by a successful CONNECT statement. Entering a program ident is analogous to connecting as a user ident, in that the program ident gains access to the system and any privileges the ident holds become applicable. Program idents are generally associated with specific functions within the system, not with physical individuals.
- **Group idents** are collective identities for groups of user or program idents. Any privileges granted to or revoked from a group ident automatically apply to all members of the group. Any ident can be a member of as many groups as required, and a group can include any number of members. Group idents provide a facility for organizing the privilege structure in the database system. All idents are automatically members of the global group ident PUBLIC.

2.1.4 Tables

Data in relational databases 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 one 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 entry in the table must have the same set of data items (i.e. columns), but not all the items need to be filled in. 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 database, which means the value is unknown, (the minus sign is an example of how the NULL value is displayed in BSQL, other applications may do it differently).

A relational database is built up of several interdependent tables that can be joined through the values in one or more columns. Thus, the GUEST_LNAME in the example here might reappear in a table of customer bills. Part of the flexibility of a relational database structure is the ability to add more tables to an existing database – for instance, a new table might be constructed for comments from guests. The new table would use the already existing GUEST_LNAME to identify guests, and no alterations of the existing data structure would be required.

All fields in any one column contain the same type of information and are of the same physical length. This length and type of information is defined in a data type. The data types supported by MIMER are (see the *MIMER/SQL Reference Manual* for a detailed description of data types):

- CHARACTER(n) - character string of fixed length n, $1 \leq n \leq 15000$
- VARCHAR(n) - character string of variable length with maximum length n, $1 \leq n \leq 15000$
- INTEGER(p) - integer number of precision p digits, $1 \leq p \leq 45$
- SMALLINT - integer number of precision 5 digits (-32768 through 32767)
- INTEGER - integer number of precision 10 digits, (-2,147,483,648 through 2,147,483,647)
- DECIMAL(p,s) - decimal number of precision p and scale s, $1 \leq p \leq 45$, $0 \leq s \leq p$ (e.g. 12.345 has precision 5 and scale 3)
- FLOAT(p) - floating point number with mantissa precision p, $1 \leq p \leq 45$ (zero or absolute value 10^{-999} to 10^{+999})
- REAL - floating point number with mantissa precision 7 (zero or absolute value 10^{-38} to 10^{+38})
- FLOAT - floating point number with mantissa precision 15 (zero or absolute value 10^{-38} to 10^{+38})
- DOUBLE PRECISION - floating point number with mantissa precision 15 (zero or absolute value 10^{-38} to 10^{+38})
- DATETIME - composed of a number of numeric fields, the value represents an absolute point in time. A variety of sub-types are supported - see the *MIMER/SQL Reference Manual* for details
- INTERVAL - composed of a number of numeric fields, the value represents a period of time. A variety of interval types are supported - see the *MIMER/SQL Reference Manual* for details

2.1.5 Base tables and views

Data in a MIMER database may be regarded as being *stored* in *base tables*: these are the tables which hold the actual database contents (although the actual physical storage form may be something other than tables). Users can directly examine data in the base tables. In addition, data may be presented in *views*, which are specific parts of one or more tables. To the user, views may appear as regular tables, but operations on views are actually performed on the underlying tables. Access privileges on views and their underlying tables are completely independent of each other.

The essential difference between a table and a view is underlined by the action of the **DROP** command, which drops 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 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 base table(s).

Note: Since views are logical representations of tables, all operations requested on a view are in fact performed on the underlying base table. For this reason, care must be taken when granting access privileges on views. These privileges may include inserting, updating and deleting information. As an example, deleting a row from a view removes the entire row from the underlying base table, including any columns which the user of the view was unable to access.

Views may be created to simplify presentation of data to the user (so-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:

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 of this kind are called *restriction views*.

Views may also be created to combine information from several tables (*join views*). Join views can be used to present data in more natural or useful combinations than the base tables themselves provide (the optimal design of base tables is 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:

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

2.1.6 Primary keys and indexes

Rows in a base table are uniquely identified by the contents of one or more columns comprising the primary key. 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 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; these columns also form a key as they may not contain duplicate values but they are not necessarily part of the primary key.

Columns in the primary key may not contain NULL (this is one of the requirements of a strictly relational database). Values in primary key columns cannot be updated – they can only be changed by deleting the row and inserting new values.

Primary keys columns are automatically indexed to aid in effective information retrieval. Other columns or combinations of columns may be defined at any time as a *secondary index* to improve performance in data retrieval; for instance if 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 result of the search is unaffected by the index. Note however that indexes cause an overhead for update, delete and insert operations. In addition, the decision whether to make use of a secondary index is made internally at the time of query execution, and there is no guarantee that an index will in fact improve performance. (SQL queries are automatically *optimized*, meaning that the database manager finds the most effective way to execute the query. In some cases, the way a query is executed internally may avoid using the index altogether.)

Indexes in MIMER are maintained automatically by the system, and cannot be accessed directly by the user.

2.1.7 Procedures

A procedure is a defined SQL routine that is stored in the data dictionary and which may be invoked when needed. For a complete discussion of the PSM functionality supported in MIMER/SQL see Chapter 8 of the *MIMER/SQL Programmer's Manual*.

MIMER/SQL supports standard procedures and also result set procedures, which are procedures capable of returning a multi-row result set.

Standard procedures are invoked directly by using the CALL statement and may return a number of single values to the caller through the procedure parameters.

In embedded SQL, result set procedures are invoked by declaring a cursor which specifies the procedure call and then by using the FETCH statement to execute the procedure and return one or more rows of a result set.

In BSQL, a result set procedure is called by using the CALL statement and the result-set values are presented in the same way as for a select returning more than one row.

The ident calling a procedure must have EXECUTE rights on it.

The creator of a procedure must hold the appropriate access rights on any database objects referenced from within the procedure. These access rights must be held for the life of the procedure.

Procedure names, like those of other database objects, can be qualified with the name of the creating ident.

The PSM constructs available in MIMER/SQL allow powerful functionality to be defined and invoked through the creation and execution of procedures. This makes it possible to move application logic from the client to the server, thereby reducing network traffic.

2.1.8 Modules

A module is simply a collection of procedures. All the procedures in a module are created when the module is created and are owned by the same ident.

EXECUTE rights on procedures in a module are held on a per-procedure basis, not on the module.

All the procedures in a module are dropped when the module is dropped.

Under certain circumstances a procedure may be dropped as a CASCADE effect of dropping some other database object (see Section 7.11.4). If such a procedure belongs to a module, the other procedures in the module remain unaffected.

In general, care should be taken when using DROP or REVOKE in connection with procedures, modules or objects referenced from within procedures (see Sections 7.11 and 8.3.4).

2.1.9 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 procedure or a module.

Using synonyms can be a convenient way to address tables created by another user. For example, if user SAMMY creates a view called ROOM_VIEW, the full name of the view is:

```
SAMMY.ROOM_VIEW
```

JIMMY may refer to this table by its full qualified name as given above. Alternatively, he may create a synonym for the view, e.g. RM_VIEW, and then simply refer to RM_VIEW. Note that the synonym RM_VIEW is owned by user JIMMY.

2.1.10 Shadows

MIMER 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. and allows backups to be taken without stopping the MIMER system. Shadowing requires a separate license, and is described in the *MIMER Shadowing Handbook*.

2.2 Data integrity

A vital aspect of a MIMER 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 functions that uphold 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 for protecting the integrity of the database, guaranteeing that undesirable data is not entered into the database. By assigning data integrity constraints to the database manager, the burden of ensuring the integrity of the database is shifted from the user to the database designer.

2.2.1 Domains

Each column in a table holds data of only one type and length, defined when the column is created. The type and length may be defined explicitly (e.g. CHARACTER(20) or INTEGER(5)), or by the use of *domains*, which can give a more precise framework for which data can be accepted in the column.

A domain definition consists of a data type and length specification with optional restriction conditions and a default value. Data which falls outside the restriction conditions is not accepted in a column defined as belonging to the domain. A column belonging to a domain for which a default value is defined is automatically assigned that value if no value is explicitly specified.

In order for an ident to create a table containing columns whose data type is defined by a domain, the ident must first have been granted USAGE rights 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 a cross-reference to the primary key or a unique key in another table. Data entered in the foreign key must either exist in the key of the referenced table or be NULL. This maintains *referential integrity* in the database, ensuring that references to non-existent data are not allowed. Conversely, a row in the referenced table cannot be deleted if the key value exists as a foreign key in another table. Foreign key relationships are defined when a table is created.

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, there cannot be a room in a hotel that doesn't exist, and a hotel cannot be deleted if it has any rooms.

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

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 B) 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 7.4.5.

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 Access rights and privileges

Access rights and privileges control users' access and operational privileges in the database.

User and program ident's 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. A password may only be changed by the ident to which it belongs or by the creator of the ident.

A set of access rights and privileges define the permitted operations for every ident in the system. There are three classes of privileges in a MIMER database:

- **System privileges**, which control the right to create new databanks and new ident's. System privileges are granted to the system administrator when the system is installed, and may be granted by the administrator to other ident's in the database. As a general rule, system privileges should be granted to a restricted group of users.
- **Object privileges**, which control membership in group ident's, the right to call procedures, the right to enter program ident's, the right to create new tables in a specified databank and the right to use a domain. 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 procedure may execute it, the creator of a program ident may enter it, the creator of a databank may create tables in the databank, and the creator of a table has all access rights on the table. The creator of an object generally has the right to grant any of these privileges to other users, in the case of procedures this actually depends on the creator's access rights on objects referenced from within the procedure.

- **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 use tables as foreign key references.

All privileges may be granted "with grant option" which means that the receiver has in turn the right to pass the privilege on to other users.

Privileges may be revoked at any time, but only by the original grantor. Grant options cannot be revoked without revoking the associated privilege. 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 categories:

- data definition statements, used to create databases

CREATE	creates objects
ALTER	modifies objects
DROP	drops objects
COMMENT	documents objects
- security control statements, used to manage access rights and privileges

GRANT	grants rights and privileges
REVOKE	revokes rights and 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 procedures
- access control statements, used to connect and disconnect user and program users to or from the database

CONNECT	connects a user user to the database
DISCONNECT	disconnects a user user from the database
SET CONNECTION	changes the active database connection
ENTER	enters a program user
LEAVE	leaves a program user
- transaction control statements, used to control when database transactions begin and end, and when updates take effect

SET TRANSACTION	set transaction modes for subsequent transactions
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 strictly part of the SQL statement syntax, but is included in the examples in this manual.

3 MANAGING DATABASE CONNECTIONS

A *database* in MIMER terms refers to the complete collection of data which may be accessed from one MIMER system. An application *connects* to a database (referred to as *logging in* in previous versions of MIMER). MIMER supports the ability to switch between different connections (i.e. access different databases) from within the same application program. A program may have several database connections open simultaneously, although only one 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` are allowed to connect to MIMER. A connection is requested from `BSQL` with the `CONNECT` statement, with the general form (see the *MIMER/SQL Reference Manual* for details):

```
CONNECT TO 'database' [AS 'connection']  
        USER 'ident' 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 System Management Handbook* for details), by specifying the database by name or by using the keyword `DEFAULT`.

Note that 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 name is given, the database name is used as the connection name.

3.1.2 Changing connections

A connection established by a successful CONNECT statement is automatically active. An application program may make multiple connections to the same or different databases using the same or different idents, provided that each connection is identified by a unique connection name. Only the most recent connection is active. Other connections are dormant, and may be made active by the SET CONNECTION statement.

```
SET CONNECTION connection;
```

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;
```

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. 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 dormant
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 dormant	con1 unaffected
DISCONNECT con2	-	con1 unaffected	con1 unaffected
SET CONNECTION con2	-	con1 made dormant	con1 unaffected

3.2 Program ids - ENTER and LEAVE

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

When a program id is entered, any privileges granted to that id become current and privileges belonging to the previous id (i.e. the id issuing the ENTER statement) are suspended.

Program ids are disconnected with the LEAVE statement.

```
ENTER 'program_1' IDENTIFIED BY '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 B).

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	GOTHEBURG

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 creator in the form *creator.table*. Unqualified table names are implicitly qualified by the user's ident name. The table name must be written in the qualified form if it is not owned by the current user, unless it is replaced by a synonym.

Example

```
SELECT *
FROM    NEWADM.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 18 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.

Certain select statements cannot be applied to views because the expansion of the statement to the corresponding base tables is illegal:

- If the view definition contains DISTINCT, you cannot then select distinct values from that view because DISTINCT may only be used once in a simple SELECT statement or subselect (see Section 4.2.4 for a description of subselects). For example, the following selection is not allowed:

```
CREATE VIEW PRICE_CODES
AS SELECT DISTINCT RESERVATION, CHARGE_CODE
FROM BILL;

SELECT DISTINCT RESERVATION
FROM PRICE_CODES;
```

The query expands to:

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

where the keyword DISTINCT appears twice.

- If the view is created with a set function (Section 4.1.6), the corresponding column in the view cannot be used in a WHERE clause:

```
CREATE VIEW BILL_TOTAL (CUSTOMER, TOTAL)
AS SELECT RESERVATION, SUM(COST)
FROM BILL
GROUP BY RESERVATION;

SELECT *
FROM BILL_TOTAL
WHERE TOTAL < 500;
```

Here the expansion is:

```
SELECT RESERVATION, SUM(COST)
FROM BILL
WHERE SUM(COST) < 500
GROUP BY RESERVATION;
```

with the illegal search condition 'WHERE SUM(COST) < 500'. (Note that in this case a HAVING clause (Section 4.1.8) may be used in place of the WHERE clause. The statement is then syntactically correct).

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, CURRENT_TIME and CURRENT_TIMESTAMP 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 clock on the 22nd of August 1997.

```
SELECT ROOMNO
FROM WAKE_UP
WHERE WAKE_DATE = DATE '1997-08-22'
AND WAKE_TIME = TIME '06:00:00';
```

ROOMNO
LAP112
SKY111
STG009

Are there any guests scheduled for checkin today, 22nd August 1997?

```
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 and NOT LIKE are used to search for character strings that match or do not 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 are only valid in LIKE statements.

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 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 "=" 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
NSSGLB	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 00:00:00';
```

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 that $AVG(column)$ is equivalent to $SUM(column)/COUNT(column)$. However, the expression $SUM(column)/COUNT(*)$ will give a different answer if the column includes NULL values. 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(AMOUNT)
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, 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 operates 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
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
SKY122	SDBLS
SKY123	SDBLS
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

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 5.5 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 'DDE' 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
DEM	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 indicator 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(AMOUNT/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	8.9342	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 5.5 of the *MIMER/SQL Reference Manual*.

Which hotel rooms have requested a wake up call within the next hour and a half (assuming current_date and time is 1997-08-22 08:35:00)?

```
SELECT ROOMNO
FROM WAKE_UP
WHERE WAKE_DATE = CURRENT_DATE
AND WAKE_TIME BETWEEN CURRENT_TIME AND
      CURRENT_TIME + 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 so far presented 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	-

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	-
TELEPHONE	-
CAR PARK	-
RESTAURANT	-
MINIBAR	-
BAR	-
ROOM SERVICE	-
LAUNDRY	-
ROOM	-
EXTRA BED	-
MISCELLANEOUS	-

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 BSQL, 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	-

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
...;
```

<i>(From CHARGES)</i>			<i>(From BILL)</i>			
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';
```

ROOMNO	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	-

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.

Note the difference in result against 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	-

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
LAP
STG
STG
STG
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 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 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' );
```

Another example illustrates that the simplest queries can be written as nested joins:

Select the names of the guests in the GUEST column of the BOOK_GUEST table.

```
SELECT GUEST_LNAME
FROM BOOK_GUEST
WHERE GUEST_LNAME IN (SELECT GUEST_LNAME
                     FROM BOOK_GUEST );
```

is equivalent to

```
SELECT GUEST_LNAME
FROM BOOK_GUEST;
```

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 HOTEL.HOTELCODE IN
      (SELECT HOTELCODE
       FROM 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 18 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 NEWADM.BOOK_GUEST.GUEST_LNAME,
       NEWADM.HOTEL.NAME, SUM(COST)
FROM   NEWADM.BOOK_GUEST, NEWADM.HOTEL, NEWADM.BILL
WHERE  NEWADM.BILL.RESERVATION = NEWADM.BOOK_GUEST.RESERVATION
AND    NEWADM.HOTEL.HOTELCODE = 'WINS'
GROUP BY NEWADM.BOOK_GUEST.GUEST_LNAME, NEWADM.HOTEL.NAME;
```

may be rewritten

```
SELECT  G.GUEST_LNAME, H.NAME, SUM(COST)
FROM    NEWADM.BOOK_GUEST AS G,
        NEWADM.HOTEL      AS H,
        NEWADM.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 subsequent references to the table in the same query must use the correlation name. The following expression is not accepted:

```
...
FROM   NEWADM.BOOK_GUEST AS G,
...
WHERE  H.RESERVATION = NEWADM.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, 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 SUBSTRING(ROOMTYPE FROM 3 FOR 4) = 'DBLS');
```

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 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 SUBSTRING (ROOMTYPE FROM 3 FOR 4) = 'SGLS' ;
```

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

```
SELECT HOTELCODE          SELECT DISTINCT HOTELCODE
FROM HOTEL                FROM ROOMS
WHERE CITY = 'STOCKHOLM' ; WHERE SUBSTRING (ROOMTYPE FROM 3
                                FOR 4) = 'SGLS' ;
```

HOTELCODE
LAP
STG

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
WIND	NSSGLB
WIND	NSSGLS
WIND	SDBLB
WIND	SSGLB
WINS	NO NSDBLB
WINS	NO NSSGLB
WINS	NO SDBLB
WINS	NO SSGLB
...	...

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
```

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.AMOUNT)
```

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;
```

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	<p>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.</p> <p>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.</p>
integer values	<p>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.</p>
floating point values	<p>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.</p>
datetime values	<p>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.</p>
interval values	<p>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.</p>

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 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 cannot be updated.

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) LIKE '% 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 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(1350, :room_no);
```

Declare a cursor which will be used when result-set data is fetched from the result 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 `WHERE` clause contains a subselect whose `FROM` clause refers to the same table or view specified in the outer select
- the `GROUP BY` clause is used in the view definition
- the `HAVING` clause is used in the view definition

6 MANAGING TRANSACTIONS

6.1 Transactions

Transactions are defined as "atomic operations", meaning groups of operations which are to be executed together (i.e. either all the statements in the transaction are executed, or none are executed).

A transaction is divided into two phases. During *build-up*, the user requests the database operations which will form the transaction. Once the build-up is complete, the transaction is *committed*, i.e. the user signals that the changes requested during transaction build-up are to be made permanent.

During build-up, changes requested in the contents of the database are not visible to other users of the system, and the database remains fully accessible to all users. Changes requested during the transaction build-up only become visible to other users when the transaction has been successfully committed.

A major function of the transaction handling in MIMER 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.

All changes made to the database within transactions may be recorded in the system logging databank, LOGDB. This contains a record of all transactions executed since the latest back-up copy of a databank was made. In the event of a databank crash, the contents of LOGDB together with the latest back-up copy of the databank may be used to restore the databank.

It is important to note that only operations included in transactions can be recorded in LOGDB.

Transaction and logging are determined on a databank basis 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;
START TRANSACTION;
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;

```

The default MIMER/SQL transaction handling settings apply to BSQL, i.e.:

```

SET TRANSACTION START IMPLICIT;
SET TRANSACTION READ WRITE;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

```

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

ALTER	ENTER	SET DATABANK	SET TRANSACTION
COMMENT	GRANT	SET DATABASE	START TRANSACTION
CREATE	LEAVE	SET SESSION	UPDATE STATISTICS
DROP	REVOKE	SET SHADOW	

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, 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 6.3.4). 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 6.3.4). 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 *MIMER/SQL Reference Manual*, Chapter 6 under 'SET TRANSACTION' 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 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, (see the *MIMER/SQL Reference Manual*, Chapter 6 under 'SET SESSION').

7 DEFINING THE DATABASE

SQL includes statements for creating and modifying the database structure:

- create idents, databanks, domains, tables, procedures, modules, views, indexes and synonyms
- saving documentary comments on objects
- altering the definition of idents, databanks 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 B. In addition, BSQL provides specific commands for listing and describing database objects (see Chapter 9).

7.1 Creating idents

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

Ident names and passwords can have a maximum length of 18 characters.

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
[IDENTIFIED BY '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 in the same way as any other user ident.

Create a user ident NEWADM with the password "Newadm".

```
CREATE IDENT NEWADM
AS USER
IDENTIFIED BY 'Newadm';
```

Create a program ident AUDIT with the password "economy".

```
CREATE IDENT AUDIT
AS PROGRAM
IDENTIFIED BY 'economy';
```

Create a group ident for the group ECONOMY_DEPT.

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

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 (which may be up to 18 characters long).
- The 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.
- The IN clause defines the file where the databank is to be stored (the form of the filename specification is machine-specific).
- The WITH clause defines the transaction handling and logging option (see Section 6.2).

Create the ROOMSDB databank with TRANS option, allocate 10 MIMER pages for it, and store it in the specified file.

```
CREATE DATABANK ROOMSDB
  OF 10 PAGES
  IN 'ROOMSDB'
  WITH TRANS OPTION;
```

At this point, the databank is empty.

7.3 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]
      [CHECK (check-condition)];
```

- The CREATE DOMAIN clause defines the domain name.
- The AS clause defines the domain data type.
- The CHECK clause defines the domain limits.

Domain names can be up to 18 characters long.

It is a good practice for upholding the integrity of the database to define domains for as many columns as possible.

7.3.1 Domains with a default value

The default clause defines values that are inserted into the column when an explicit value is not specified 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(18)
      DEFAULT USER;
```

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 outside the check limits, an explicit value must always be inserted into the column.

7.3.2 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 5.9 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 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');
```

7.4 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 18 characters long.

As a convention, we have defined primary key column(s) as the first column(s) in the table definition. 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 BOOKDB databank.

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

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 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
- columns not belonging to the primary key defined as NOT NULL
- unique columns (in addition to the primary key)
- default values (overriding any domain default for the column)
- foreign key references
- check conditions

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),
                          BOOKING_DATE  DATE          NOT NULL,
                          HOTELCODE     HOTELCODE    NOT NULL,
                          ROOMTYPE      ROOMTYPE     NOT NULL,
                          RESERVED_BY   PERSONNAME   NOT NULL,
                          TELEPHONE     CHAR(15),
                          RESERVED_FOR  PERSONNAME,
                          ARRIVE         DATE          NOT NULL,
                          DEPART        DATE          NOT NULL,
                          GUEST         PERSONNAME,
                          ADDRESS       CHAR(30),
                          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 ROOMSDB;
```

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),
   PRIMARY KEY (RESERVATION),
   BOOKING_DATE DATE          NOT NULL,
   HOTELCODE     HOTELCODE    NOT NULL,
   FOREIGN KEY (HOTELCODE) REFERENCES HOTEL,
   ROOMTYPE      ROOMTYPE     NOT NULL,
   FOREIGN KEY (ROOMTYPE) REFERENCES ROOMTYPES,
   ...
```

7.4.1 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.

7.4.2 The primary key

The primary key 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.

7.4.3 Alternate key

An alternate key can consist of one or more columns in the table. The list of columns that make up the alternate key are specified in the UNIQUE clause for the table when it is created.

This is the recommended way of defining an alternate key, 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 alternate key.

Creating a UNIQUE index on the table has the same effect as an alternate key.

7.4.4 Foreign keys - referential integrity

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

Note that 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).

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 unique key columns 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: Neither foreign keys nor tables referenced in foreign keys may be included in databanks that are defined with the NULL option.

The BOOK_GUEST table has three foreign key references:

```
CREATE TABLE BOOK_GUEST (RESERVATION    INTEGER(5) ,
                          BOOKING_DATE  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 )
                          .
```

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.

7.4.5 Check conditions

Check conditions 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 conditions.

Note that the first two examples 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),
                    CHECK (CITY IN ('STOCKHOLM', 'GOTHENBURG')))
IN ROOMSDB;
```

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 ROOMSDB;
```

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.

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 BOOKDB;
```

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).

7.5 Creating procedures and modules

Procedures are SQL routines that are stored in the data dictionary. A **module** is a collection of procedures.

A module is created by using the CREATE MODULE statement and all the procedures that belong to the module are defined by declaring them within the CREATE MODULE statement.

Procedures cannot be added to a module after the module has been created and a procedure cannot be removed from the module it belongs to. The procedures in a module behave in all respects as single objects (e.g. EXECUTE privilege is applied on individual procedures in a module, not the module). If the module is dropped, all the procedures in it are dropped.

The CREATE PROCEDURE statement is used to create a procedure that does not belong to a module. The format of the procedure definition is the same in the CREATE PROCEDURE statement as it is in a procedure declaration in a module.

Refer to the *MIMER/SQL Reference Manual* for the syntax definitions for CREATE MODULE and CREATE PROCEDURE, 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.

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) and PROC_2 (one input parameter, X, of data type 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;
END MODULE
@
```

Note: It is recommended that all procedures 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 procedures and modules are concerned (see Section 8.8 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 for it in the command file and creating the new, restructured module.

7.6 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 persons 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 18 characters long. Views are defined in terms of a SELECT statement; the result of the SELECT statement forms the contents of the view.

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, 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
1348	40	TELEPHONE
1348	250	RESTAURANT
1348	200	BAR
1350	100	LODGING
1350	70	MINIBAR
1350	100	LODGING
1350	250	RESTAURANT
1350	200	BAR
1350	120	LAUNDRY
1350	100	LODGING
...
...

7.6.1 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 in to 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.

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;
```

7.7 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. Note however that columns in the primary key, columns addressed through a FOREIGN KEY reference from another table and columns defined as UNIQUE are automatically indexed, (in the order in which they are defined in the key), and 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 18 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 in 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, AD 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 searches indexes forwards or backwards depending on the circumstances.

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

7.8 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 18 characters.

Table names are "qualified" by the name of their creator. The qualified form of the table name is the creator name followed by the table name and the two are separated by a period. Thus the table ROOMS with the creator NEWADM has the qualified name:

```
NEWADM.ROOMS
```

The table's creator need only refer to it as:

```
ROOMS
```

Its qualifier is implicit since the creator is using the table. However, should another user wish to use this table, he must refer to it by its full qualified name since he is not its creator.

If a user named James wishes to refer to the ROOMS table belonging to the user NEWADM as simply ROOMS, he can create a synonym:

```
CREATE SYNONYM ROOMS  
FOR NEWADM.ROOMS;
```

Another user can then create his own synonym for James' ROOMS synonym, which has the full name:

```
JAMES.ROOMS
```

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

7.9 Commenting objects

Comments may be stored against any of the following objects:

COLUMN	IDENT	PROCEDURE	TABLE
DATABANK	INDEX	SHADOW	VIEW
DOMAIN	MODULE	SYNONYM	

Store the comment "MIMER Hotels Databank" on the BOOKDB databank.

```
COMMENT ON DATABANK BOOKDB 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 an object 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 data dictionary tables.

7.10 Altering databanks, tables and idents

7.10.1 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 BOOKDB is stored in from its previous file to file "SQLDB:NEWDB.DBF" (the file specification is in VAX/VMS format).

```
ALTER DATABANK BOOKDB
  INTO 'SQLDB:NEWDB.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 BOOKDB databank from TRANS to LOG.

```
ALTER DATABANK BOOKDB
  TO LOG OPTION;
```

Increase the size of the BOOKDB database by 20 MIMER pages.

```
ALTER DATABANK BOOKDB
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.

7.10.2 Altering tables

The ALTER TABLE statement changes the definition of the specified table and has no effect on the table's existing contents. Only the creator of the table may alter it.

There are four uses for the ALTER TABLE statement:

- to add a new column to an existing table
- to drop a column from an existing table
- to change the default value for a column in an existing table
- to drop the default value for a column in 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, if no default value exists, the NULL indicator.

A column added to an existing table has the following limitations:

- it cannot be defined as a primary, unique or foreign key
- it cannot include a CHECK clause
- it cannot include a default value (unless column defined as a domain)
- it can only be NOT NULL if it belongs to a domain with 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.

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 (the default) is specified, depending objects are also dropped. For instance if a dropped column is part of a secondary index, the index will also be dropped. If RESTRICT 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;
```

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 default value for the column BOOKING_DATE.

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

7.10.3 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  
IDENTIFIED BY 'SamJo';
```

7.10.4 Objects which may not be altered

Domains, procedures, modules, 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 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 procedures contained in it can, of course, be dropped).

The next section will discuss dropping objects and the results of this on the database.

7.11 Dropping objects from the database

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

DATABANK	INDEX	SHADOW	VIEW
DOMAIN	MODULE	SYNONYM	
IDENT	PROCEDURE	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 is specified, an error is returned if other objects are affected, and the drop operation is aborted. If CASCADE (the default) is specified, dependent objects are dropped as well. Objects can only be dropped by their creator.

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, and indexes based on 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 7.10.2).

7.11.1 Dropping databanks and tables

Drop the HOTEL table.

```
DROP TABLE HOTEL;
```

Any views, synonyms and indexes based on HOTEL are also dropped as well as any procedures referencing the table.

Drop the BOOKDB databank.

```
DROP DATABANK BOOKDB;
```

All tables in the BOOKDB databank are also dropped and any views, synonyms and indexes based on those tables are also dropped as well as any procedures 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, that the database server's attempt to delete the physical databank file might, however, fail. If recommended procedures for databank file management are followed (see the *MIMER System Management Handbook*), the databank file should be deleted correctly.

7.11.2 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;
```

Note that if you re-create a domain that has been dropped, the domain will be seen as being a completely new domain and it will not be tied to any columns that previously belonged to the old domain.

To change the restrictions on those columns that were defined with a domain that has been dropped, use the LOAD and UNLOAD utilities described in Chapter 9. The procedure to be followed is listed below:

1. Unload the data in the table into a sequential file with the UNLOAD utility.
2. Drop the table from the databank.
3. Re-create the table.
4. Load the data with the LOAD utility from the sequential file containing the old table into the new table.

7.11.3 Dropping idents

When an ident is dropped, everything that the ident has created (including 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;
```

7.11.4 Dropping procedures and modules

The effect of CASCADE can be rather dramatic when procedures (and modules) are dropped. It is for this reason that it is recommended that all modules and procedures be created by running a **command file** so they can be easily reconstructed in case of being dropped in error.

Drop the procedure called ADD_LODGING.

```
DROP PROCEDURE ADD_LODGING;
```

Drop the module called ROOMS_ADMIN.

```
DROP MODULE ROOMS_ADMIN;
```

The following points should be noted when dropping procedures and modules:

- When a module is dropped, all the procedures contained in it will be dropped (this is not a cascade effect, but it may provoke cascade effects).
- If a procedure is dropped and it is called from another procedure, the calling procedure will also be dropped.
- If a procedure references a database object (table, view etc.) and the referenced object is dropped, the procedure will also be dropped.
- In order for a procedure to reference a database object, the creator of the procedure must hold appropriate access rights on the referenced object. If those access rights are revoked from the creator of such a procedure, the procedure will be dropped.
- If a procedure belonging to a module is to be dropped as a consequence of a cascade, only that procedure is dropped (the other procedures in the module and the module itself will remain unaffected).

8 DEFINING PRIVILEGES

Privileges and access rights control the operations which users are allowed to perform in the database. Well-structured privileges and access rights 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 system.
- Access privileges, which give rights of access to the data in a specified table.

System privileges are granted to the system administrator upon installation, and may be passed on to other users. Objects and access privileges are initially granted only to the creator of an object. The creator may however pass the privileges on to other users.

Privileges are granted to users with the GRANT statement and revoked from users with the REVOKE statement.

All privileges may be granted "with grant option", which means that the receiver of the privilege in turn has the right to grant that privilege to other users.

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 user may enter it, and the creator of a table has full access privileges, etc.

When privileges that were granted "with grant option" are revoked, the right to grant those privileges to other users is also revoked. Grant option cannot be revoked without revoking the privilege. Users may only grant privileges that they themselves possess to other users, that is, users cannot grant privileges to themselves. Likewise, privileges may only be revoked by the grantor - users 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 USER, which may be performed by either the user himself or by the creator of the user), 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 DATABANK and IDENT privileges with GRANT OPTION, and SELECT access on all tables 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 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 EXECUTE privilege on program idents MIMER_BR and MIMER_SC (see the *MIMER System Management Handbook*).

The initial installation of MIMER also includes a general group ident called PUBLIC. All idents in the system are automatically members of the group PUBLIC, which may thus be used for granting global privileges.

The following general recommendations can be made for structuring the idents in a system:

- Roles within the system 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).
- Physical users of the system are 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 added 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.
- Physical user idents should not in general be allowed to create objects (i.e. granted DATABANK, IDENT or TABLE privileges). In this way, individual user idents may be dropped with no cascading effects except loss of views created by the user.
- GRANT OPTION should be used sparingly and the ident hierarchy kept shallow. This minimizes the risk of cascading revocation of privileges.

Give ECONOMY_DEPT the privilege to enter the AUDIT program ident.

```
GRANT EXECUTE ON 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 ECONOMY_DEPT the privilege to create new tables in the BOOKDB databank.

```
GRANT TABLE ON BOOKDB
      TO ECONOMY_DEPT;
```

Give ECONOMY_DEPT 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.
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 NEWADM 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 NEWADM
      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 not belonging to the primary key in the BOOK_GUEST table.

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

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

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

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

```
GRANT REFERENCES
  ON NEWADM.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 a cascading effect on all idents who have been granted privileges by that ident (see Section 8.3.4).

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 individual's membership in the group.

8.3.1 Revoking system privileges

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

```
REVOKE DATABANK
  FROM NEWADM;
```

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

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

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;
```

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

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

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

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

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

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

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 retrieve data to and from the BOOK_GUEST table from the ident MARIANNE.

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

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;
```

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

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.

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 PROCEDURE). The use of '@' is not intended 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 that for unattended operation, a batch file must either include the MIMER 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 on the platform you are using, refer to the set of *MIMER Guides* supplied for your specific MIMER installation. Starting BSQL displays the following screen:

```

MMMMM      MMMMM  MMMMM  MMMMM      MMMMM  MMMMMMMMMMMM  MMMMMMMMM
MMMMMMM    MMMMMM  MMMMM  MMMMMM    MMMMMM  MMMMMMMMMMMM  MMMMMMMMMMM
MMMMMMM    MMMMMM  MMM    MMMMMM  MMMMMM    MMM  MMM    MMM  MMM
MMMMMMMMMMMMMMMM  MMM    MMMMMMMMMMMMMMMM  MMMMM    MMMMMMMM
MMM  MMMMM  MMM  MMM    MMM  MMMMM  MMM  MMM  MMM  MMM  MMM
MMMM  MMM  MMMM  MMMMM  MMMM  MMM  MMMM  MMMMMMMMMMMM  MMMM  MMMM
MMMM  M  MMMM  MMMMM  MMMM  M  MMMM  MMMMMMMMMMMM  MMMM  MMMM

```

(C) Copyright SYSDECO MIMER AB 1987-1997. All rights reserved

M I M E R / B S Q L
Version x.x.x

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.

9.2 BSQL commands

Command	Function
CLOSE	Close active log files
DESCRIBE	Describes a specified object
EXIT	Leaves BSQL
HELP	Provides on-line help
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          4. Index          7. View
2. Domain            5. Synonym         8. Module
3. Ident             6. Table           9. Procedure
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 idents with table privilege on the databank.
	FULL	Lists the following information on the specified databank: creator file space used allocated size physical file name option tables a list of all the idents with table privilege on the databank comments creation date.
DOMAIN	BRIEF	Lists the following information on the specified domain: creator data type default value check clause.
	BY TABLES USING	Lists the following information on the specified domain: tables using domain columns in those tables that use the domain in their definition.
	BY PROCEDURES USING	Lists the procedures using the domain, i.e. in a CAST operation.
	BY USAGE PRIVILEGE	Lists the idents that have usage privilege on the domain.
	FULL	Lists the following information on the specified domain: creator data type default value check clause tables using domain idents with usage privilege on the domain procedures using the domain in a CAST columns in those tables that use the domain in their definition comments creation date.

DESCRIBE	OPTION	RESULT
IDENT	BRIEF	Lists the following information on the specified ident: creator type of ident (user, os_user, program or group) privileges held by ident (DATABANK or IDENT).
	BY ACCESS	Lists the following information on the specified ident: tables and columns to which the ident has access access privileges that the ident has on the tables and columns listed above.
	BY MEMBERSHIP	Lists groups of which the ident is a member.
	BY EXECUTE ON PROGRAM	Lists programs on which the ident has the execute privilege.
	BY EXECUTE ON PROCEDURE	Lists procedures on which the ident has the execute privilege.
	BY OWNERSHIP	Lists objects that the ident has created: object type object name.
	BY TABLE PRIVILEGE	Lists databanks in which the ident has the right to create tables.
	BY USAGE PRIVILEGE	Lists domains on which the ident has usage privilege.
	FULL	Lists the following information on the specified ident: creator ident type privileges objects to which the ident has access program names on which the ident has execute privilege procedures on which the ident has execute privilege group names (of which the ident is a member) objects which the ident has created databanks on which the ident has table privileges comments creation date domains on which the ident has usage privilege.

DESCRIBE	OPTION	RESULT
INDEX	BRIEF	Lists the following information on the index: whether index is unique creator index name table name columns on which the index is defined whether sort order is descending comments creation date
SYNONYM	BRIEF	Lists the following information on the specified synonym: creator original name of the table/view creator of the table/view
	BY ACCESS	Describes the table with the specified synonym name and which idents have access to that synonym.
	FULL	Lists the following information on the specified synonym: creator table/view name table/view creator idents with access to synonym comments creation date

DESCRIBE	OPTION	RESULT
TABLE	VERY BRIEF	Lists the column names and data types in a table, view or synonym.
	BRIEF	Lists the following information on the table/view/synonym: creator column names (data type, size, scale) primary key default values unique constraint domains (used in column definitions) indexes
	BY ACCESS	Lists which idents have access to the table/view/synonym: ident names access privileges
	BY FOREIGN KEYS	Describes which foreign key references the table/view/synonym contains: referencing table referencing columns referenced table referenced columns
	BY VIEWS	Lists views defined on the table/view/synonym.
	BY PROCEDURES	Lists the procedures which contain references to the table.
	FULL	Lists the following information on the specified table/view/synonym: creator column names (data type, size, scale) primary key default values unique constraint domains used indexes idents with access to table foreign key references procedures referencing the table views defined on table comments creation date date when statistics were generated

DESCRIBE	OPTION	RESULT
VIEW	BRIEF	Lists the following information on the specified view: creator view definition
	BY ACCESS	Lists the idents that have access privileges on the view.
	FULL	Lists the following information on the specified view: creator view definition ident names with access to the view comments creation date
MODULE	BRIEF	Lists the source definition for the module.
PROCEDURE	BRIEF	Lists the following information on the specified procedure: procedure name creator list of parameters showing name, mode and data type list of result items showing name and data type.
	BY ACCESS	Lists the idents that have execute privilege on the specified procedure.
	BY REFERENCES	Lists the objects referenced in the procedure, showing creator, object name, column name (if any), type of object, access privilege used.
	FULL	Lists the following information on the specified procedure: procedure name creator list of parameters showing name, mode and data type list of result items showing name and data type idents that have execute privilege on it list of objects referenced from the procedure source definition for the procedure module name the procedure belongs to.

EXIT

Leave BSQL.

Syntax

EXIT;

Description

Leave BSQL and return to the operating system.

HELP

Provides help text on the specified argument.

Syntax

HELP [argument];

Description

Provides a general help text or help text on the specified argument. The help text is displayed one screen at a time according to the screen length set by the SET LINECOUNT command. Press RETURN to display the next screen. If LINECOUNT is set to zero, the help text will be shown without interruption.

At the end of the help text for a specified argument, a menu lists related topics and offers the two additional choices CONTENTS and QUIT. CONTENTS lists all topics available in the help system. QUIT leaves the help system.

LIST

Lists information on a specified object.

Syntax

LIST [object-type];

Description

The LIST command presents the following menu:

```

Menu for List

1. Databanks      5. Objects      9. Modules
2. Domains        6. Synonyms    10. Procedures
3. Idents         7. Tables
4. Indexes        8. Views       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.
	WITH OPTION	Lists databanks with a specified option (LOG, TRANS or NULL).
	ALL SHADOWS	Lists all shadows in the database.
	SHADOWS	Lists all shadows for a specified databank.
DOMAINS	DOMAINS	Lists all domains in the database.
	CREATED BY	Lists domains created by a specified ident.
	USED IN PROCEDURE	Lists domains used (in CAST operations) in the specified procedure.

LIST	OPTION	RESULT
IDENTS	ALL	Lists all idents in the database.
	CREATED BY	Lists idents created by a specified ident.
	WITH TYPE	Lists idents of a specified type (PROGRAM, USER, OS_USER or GROUP).
	WITH ACCESS ON	Lists idents that have access to a specified table.
	WITH EXECUTE ON	Lists idents that have EXECUTE privilege on a specified program.
	WITH MEMBER ON	Lists idents that are members of a specified group.
	WITH TABLE PRIVILEGE ON	Lists idents that have TABLE privilege on a specified databank.
	WITH DATABANK PRIVILEGE	Lists idents with DATABANK privilege.
	WITH IDENT PRIVILEGE	Lists idents with IDENT privilege.
	WITH RESTRICTED UPDATE ON	Lists idents with restricted UPDATE privilege.
	WITH EXECUTE ON PROCEDURE	Lists idents that have EXECUTE privilege on a specified procedure.
	WITH USAGE ON DOMAIN	Lists idents that have USAGE privilege on a specified domain.
INDEXES	ALL	Lists the secondary indexes in the database.
	CREATED BY	Lists secondary indexes created by a specified ident.
	DEFINED ON TABLE	Lists secondary indexes defined on a specified table.
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.
	DEFINED ON TABLE / VIEW	Lists synonyms defined for a specified table, view or synonym.

LIST	OPTION	RESULT
TABLES	ALL	Lists tables in the database.
	CREATED BY	Lists tables created by a specified ident.
	USING DOMAIN	Lists tables that use a specified domain in the table definition.
	IN DATABANK	Lists tables in a specified databank.
	USED IN PROCEDURE	Lists tables referenced in the specified procedure.
VIEWS	ALL	Lists views in the database.
	BY CREATOR	Lists views created by a specified ident.
	DEFINED ON TABLE / VIEW	Lists views defined on a specified table.
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.
	USING TABLE	Lists procedures containing references to the specified table.
	USING PROCEDURE	Lists procedures containing calls to the specified procedure.
	USING DOMAIN	Lists procedures using (in CAST operations) the specified domain.

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|NONNULL,>
      <DUPLICATES|NODUPLICATES> <LOGFILE 'file-name'>
      < (column-name POS(s:e), ..., column-name POS(s:e)) >;
```

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. If a LOGFILE is specified, any duplicate row will also be logged there. NODUPLICATES means that number of duplicates will not be reported or logged.

LOGFILE specifies a sequential file, where duplicate rows 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 column-specifications are not given, 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));
```

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 and TRANSACTION START.

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|NONNULL>
      <(column-name POS(s:e), ..., column-name POS(s:e)) >;
```

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 column-specifications are not 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));
```

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
```

In the first construction, the host identifier is the name of the main host variable. In the second construction, 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, MIMER 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';
```

Note that 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: 1997-04-23
Null:Y
```

Note that 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 1997-04-23 and the DEPART value will be set to NULL.

The value prompting makes host variables limited to 80 characters in BSQL.

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)
        ^
Invalid operand type, expected type is NUMERIC or INTERVAL
FROM HOTEL;
```

```
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
-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 MACHINE-DEPENDENT INFORMATION

The information in this manual is, as far as possible, independent of the platform on which MIMER/SQL is used. While the MIMER product is designed to be identical on all platforms, there are some inevitable differences in the user interface on different platforms. Any platform-dependent information is provided in the set of *MIMER Guides* supplied for the platform you are using.

If you do not have a copy of the *MIMER Guides* for your particular platform, contact your system administrator or equivalent person.

B EXAMPLE DATABASE

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

The database consists of two databanks, BOOKDB and ROOMSDB. The BOOKDB databank contains information needed when booking guests: information on guests, available rooms and room status. The ROOMSDB databank contains information on the hotels: hotel locations, room types, number of rooms per hotel, prices, etc.

All tables in the example database are created by the ident NEWADM.

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 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.

B.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.

B.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		<i>f</i>	CHAR(4)	Hotel identity code
ROOMTYPE		<i>f</i>	CHAR(6)	Room type
STATUS		<i>f</i>	CHAR(10)	Room status

ROOM_PRICES				
HOTELCODE	*	<i>f</i>	CHAR(4)	Hotel identity code
ROOMTYPE	*	<i>f</i>	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		<i>f</i>	CHAR(4)	Hotel identity code
ROOMTYPE		<i>f</i>	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		<i>f</i>	CHAR(7)	Room number
PAYMENT			CHAR(10)	Payment type

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

WAKE_UP				
ROOMNO	*	<i>f</i>	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

B.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
WIND	WINSTON	COPENHAGEN
WINS	WINSTON	GOTHENBURG
WIN	Winston	London

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	SYSDECO MIMER 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

B.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 NEWDB
  OF 60 PAGES
  IN 'NEWDB'
  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 NEWDB;
```

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

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

```
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 NEWDB;
```

```

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 NEWDB;

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

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 NEWDB;

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 NEWDB;

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 NEWDB;

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

```

```

--
-- 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 CURRENT_TIMESTAMP
    AND CURRENT_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,
             CURRENT_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
@

```

INDEX

A

- access rights 2-11, 8-1
- active connection 3-2
- ALL 4-33
- ALTER DATABANK 7-14
- ALTER IDENT 7-16
- ALTER TABLE 7-15
- ANY 4-33
- arithmetic operations 4-10
- AS
 - for column labels 4-3
 - for connection name 3-1
- AVG 4-11

B

- base tables 2-5
- batch operation 9-1
- BETWEEN condition 4-8
- BSQL
 - Commands 9-1
 - Errors 11-1

C

- CASCADE 7-17, 8-7
- CASE 4-18
- CAST 4-19
- changing connections 3-2
- changing passwords 7-16
- CHAR_LENGTH 4-16
- character set 4-5
- character string comparison 4-5
- CHECK
 - in domain 7-4
- check conditions 2-10
- check conditions in tables 7-8
- check option in views 2-11, 7-11
- client/server 2-1
- CLOSE
 - BSQL 9-4
- column labels 4-3
- column names in UNION 4-35
- comments 7-14
- COMMIT 9-21
- comparison 4-5
- computed values 4-10
- CONNECT 3-1

- connections 3-1
- CONTINUE 9-21
- correlation names 4-29
- COUNT 4-11
- creating
 - databanks 7-2
 - domains 7-3
 - idents 7-1
 - modules 7-9
 - procedures 7-9
 - secondary indexes 7-12
 - synonyms 7-13
 - tables 7-4
 - views 7-10
- cross product 4-22
- D**
- data dictionary 2-1
- data integrity 2-9
- data manipulation 5-1
- data types 2-4
- databank 2-2
- databank options 6-2
- DATABANK privilege 8-3
- databank shadows 2-8
- databanks
 - altering 7-14
 - creating 7-2
 - dropping 7-17
- database connections 3-1
- database definition statements 7-1
- database design 7-1
- database organization 2-1
- datetime arithmetic 4-19
- datetime functions 4-19
- default values in domains 7-3
- DELETE 5-5
- DELETE access 8-4
- delimiting complex statements with '@' 7-9, 9-1
- DESCRIBE
 - BSQL 9-4
- DISCONNECT 3-2
- DISTINCT 4-3
 - in set functions 4-11
- domains 2-9
 - creating 7-3
 - default values 7-3
 - dropping 7-18
- dropping objects 7-17
- duplicate values 4-3
- E**
- ECHO 9-16
- embedded SQL 1-1
- ENTER 3-3
- error handling 11-1
- ESCAPE in LIKE conditions 4-7
- ESQL 1-1

- example database B-1
- EXECUTE privilege 8-3
- EXISTS
 - NULL values 4-38
- EXISTS condition 4-31
- EXIT
 - BSQL 9-10, 9-21
- EXTRACT 4-16
- F**
- FORALL 4-32
- foreign keys 2-9, 7-6
- G**
- grant option 2-12, 8-1
- granting privileges 8-3
- GROUP BY 4-13
- group idents 2-3
- H**
- HAVING 4-14
- HELP
 - BSQL 9-10
- host variables 10-1
- I**
- IDENT privilege 8-3
- idents 2-2
 - altering 7-16
 - creating 7-1
 - dropping 7-18
 - organization 8-2
- IN condition 4-8
- indexes 2-6
- indicator variables 10-1
- INSERT 5-1
- INSERT access 8-4
- inserting NULL values 5-4
- inserting with a subselect 5-3
- inserting with a values list 5-3
- IS NULL 4-37
- isolation levels in transactions 6-3
- J**
- join condition 4-21
- join views 2-5
- joining a table with itself 4-30
- joins
 - natural 4-23
 - outer 4-25
 - simple 4-23
- K**
- keys 2-6
- L**
- LEAVE 3-3
- LIKE 4-6
- LINECOUNT 9-16
- LINESPACE 9-17

- LINEWIDTH 9-17
- LIST
 - BSQL 9-11
- LOAD
 - BSQL 9-14
- LOG 9-17
 - BSQL 9-15
- LOG databank option 6-2
- LOGDB 2-2, 6-1
- logging 6-1
- logical operators 4-5
- LOWER 4-16
- M**
- MAX 4-11
- MEMBER privilege 8-3
- merging with UNION 4-35
- MESSAGE 9-18
- MIN 4-11
- modules 2-7
 - creating 7-9
- N**
- natural joins 4-23
- nested selects 4-26
- NULL databank option 6-2
- NULL values
 - in EXISTS etc. 4-38
 - in SELECT 4-37
 - in set functions 4-11
 - in variables 10-1
 - inserting 5-4
 - treated as equal by distinct 4-4
- O**
- object names 2-2
- object privileges 2-11
- objects 2-1
- optimization 2-6
- optimizing transactions
 - READ ONLY and READ WRITE 6-3
- ORDER BY 4-15
 - in subselects 4-28
- OS_USER 2-3
- outer joins 4-25
- outer references 4-30
- OUTPUT 9-18
- P**
- PAGELength 9-18
- PAGEWIDTH 9-19
- passwords 7-1
- pattern conditions 4-6
- POSITION 4-16
- primary key 2-6, 7-6
- private objects 2-2
- privileges 2-11, 8-1
- procedures 2-7
 - creating 7-9

- procedures and modules
 - protection against CASCADE effects 7-19
- program idents 2-3, 3-3
- PUBLIC group ident 8-2

Q

- quantified predicates 4-33

R

- READ INPUT 9-15
- recursive effects of revoking privileges 8-7
- REFERENCES 7-6
- REFERENCES access 8-4
- referential integrity 2-9, 7-6
- RESTRICT 7-17, 8-7
- restriction views 2-5
- result table 4-1
- retrieving data
 - from multiple tables 4-21
 - from single tables 4-1
- revoking privileges 8-5
- ROLLBACK 9-21

S

- scalar functions 4-16
- searching for NULL 4-37
- secondary indexes 2-6
 - creating 7-12
- SELECT
 - computed values 4-10
 - creating views 7-10
 - DISTINCT 4-3
 - EXISTS 4-31
 - GROUP BY 4-13
 - HAVING 4-14
 - illegal statements on views 4-4
 - NULL values 4-37
 - ordering the result 4-15
 - quantified predicate 4-33
 - simple form 4-1
 - UNION 4-35
 - WHERE 4-5
- SELECT access 8-4
- selecting groups 4-14
- selection process 4-40
- semantic errors 11-1
- sequential command files 9-15
- SET
 - CONNECTION 3-2
 - ECHO 9-16
 - LINECOUNT 9-16
 - LINESPACE 9-17
 - LINEWIDTH 9-17
 - LOG 9-17
 - MESSAGE 9-18
 - OUTPUT 9-18
 - PAGELength 9-18
 - PAGEWIDTH 9-19
- set conditions 4-8

- set functions 4-11
- SET SESSION 6-4
- SET TRANSACTION 6-2
- SETTINGS 9-19
- shadowing 2-8
- SHOW SETTINGS 9-19
- simple joins 4-23
- SOME 4-33
- source table 4-1
- SQL statements 2-12
- stored procedures 2-7
- string concatenation 4-10
- subselects 4-26
 - in INSERT 5-3
- SUBSTRING 4-16
- SUM 4-11
- synonyms 2-8
 - creating 7-13
- syntax errors 13- 11-2
- SYSADM 8-2
- SYSDB 2-2
- system databanks 2-2
- system objects 2-2
- system privileges 2-11
- system utilities 8-2

T

- TABLE privilege 8-3
- tables 2-3
 - altering 7-15
 - check conditions 7-8
 - column definitions 7-6
 - creating 7-4
 - dropping 7-17
- TRANS databank option 6-2
- transaction consistency
 - SET TRANSACTION ISOLATION LEVEL 6-3
- transaction control options - setting defaults 6-4
- transaction control statements 6-2
- transaction optimization 6-3
- transactions 6-1
- TRANSDB 2-2
- TRIM 4-16

U

- UNION 4-34
- UNLOAD
 - BSQL 9-20
- updatable views 5-6
- UPDATE 5-4
 - UPDATE access 8-4
- UPPER 4-16
- USAGE ON DOMAIN privilege 8-3
- user databanks 2-2
- user idents 2-2

V

variables 10-1

views 2-5

check option 7-11

check options 2-11

creating 7-10

updatable 5-6

W

WHENEVER

BSQL 9-21

WHERE condition 4-5

wildcard characters 4-6