

# ADABAS D TUTORIAL



**Manual Order Number: ESD611-021ALL**

This document is applicable to ADABAS D Version 6.1.1 PE and to all subsequent releases, unless otherwise indicated in new editions or technical newsletters.

Specifications contained herein are subject to change and these changes will be reported in subsequent revisions or editions.

Readers' comments are welcomed. Comments may be addressed to the Documentation Department at the address on the back cover.

© April 14, 2025, SOFTWARE AG, Germany & SOFTWARE AG of North America, Inc.

All rights reserved

Printed in the Federal Republic of Germany

The SOFTWARE AG documentation often refers to numerous hardware and software products by their trade names. In most, if not all cases, these designations are claimed as trademarks or registered trademarks by their respective companies.

# 1 Introduction

ADABAS D is a relational database system with an SQL (Structured Query Language) compatible language interface. ADABAS data is organized in the form of tables. A set of simple statements based on the English language serves to formulate a large variety of database operations.

This means specifically:

The SELECT statement enables the user to retrieve data from a table row by row or column by column. Rows from several tables can be combined with each other. Data can be sorted, grouped, and used for calculations.

Rows are inserted using the INSERT statement and deleted using the DELETE statement. The UPDATE statement can be used to modify the data.

Another group of statements serves to define and redefine the structure of a table.

We recommend the reader to work directly with the database by trying out the examples provided in this book. For certain problems, you may thus approach the result step by step. You may build your own samples for this purpose or use existing examples.

With each ADABAS version, demo samples are distributed which can be loaded automatically. Instructions for their installation are contained in the corresponding README file. The sample tables mainly correspond to the tables described here and can therefore help you to learn on the system.

This tutorial makes the reader acquainted with the SQL language on the basis of the database system ADABAS. The manual is both a general and a special text book that also explains some of the ADABAS features that exceed the SQL standard.

The following standard literature is recommended as a general introduction to the SQL language:

- (1) LAN Times Guide to SQL  
James R. Groff, Paul N. Weinberg  
McGraw-Hill, 1994
- (2) Introduction to Database System  
C.J. Date  
Addison Wesley, 1990

## 2Fundamentals

The following examples are based on a table called 'customer'. It contains the following data:

CNO	TITLE	NAME	FIRSTNAME	CITY	STATE	ZIP	ACCOUNT
3000	Mrs	Porter	Jenny	New York	NY	10580	100.00
3100	Comp	DATASOFT	?	Dallas	TX	75243	4813.50
3200	Mr	Randolph	Martin	Los Angeles	CA	90018	0.00
3300	Mrs	Smith	Sally	Los Angeles	CA	90011	0.00
3400	Mr	Brown	Peter	Hollywood	CA	90029	0.00
3500	Mr	Jackson	Michael	Washington	DC	20037	0.00
3600	Mr	Howe	George	New York	NY	10019	-315.40
3700	Mr	Miller	Frank	Chicago	IL	60601	0.00
3800	Mr	Peters	Joseph	Los Angeles	CA	90013	650.00
3900	Mrs	Baker	Susan	Los Angeles	CA	90008	-4167.79
4000	Mr	Jenkins	Anthony	Los Angeles	CA	90005	0.00
4100	Mr	Adams	Thomas	Los Angeles	CA	90014	-416.88
4200	Mr	Griffith	Mark	New York	NY	10575	0.00
4300	Comp	TOOLware	?	Los Angeles	CA	90002	3770.50
4400	Mrs	Brown	Rose	Hollywood	CA	90025	440.00

In this simple data model, each row contains the complete information stored about a particular customer: a number for clear identification, the title to be used in correspondence, the name, the address, and the current account. Other columns needed in real applications have been omitted for the sake of clarity.

From this table, which can become very long and very wide, subtables can be selected for processing purposes or for special tasks. This can be done very easily with the following method:

## 2.1 Example of a SELECT Statement

For clarity reasons, UPPER case characters are used in the following for ADABAS statements, lower case characters for table and column names.

```
SELECT title, name, city, state, zip
      FROM customer
        WHERE state = 'CA'
          ORDER BY name
```

This means:

- \* Find (SELECT)
- \* the columns title, name, city, state, zip
- \* in the table 'customer' (FROM)
- \* for the rows with the value 'CA' in the column 'state' (WHERE)
- \* and order the result rows alphabetically according to name (ORDER BY)

The general form of a SELECT statement looks like this:

```
SELECT.....  which columns
FROM.....     which table
WHERE.....     the condition is
ORDER BY...    in the following way
```

The result of the SELECT statement is the following table:

TITLE	NAME	CITY	STATE	ZIP
Mr	Adams	Los Angeles	CA	90014
Mrs	Baker	Los Angeles	CA	90008
Mr	Brown	Hollywood	CA	90029
Mrs	Brown	Hollywood	CA	90025
Mr	Jenkins	Los Angeles	CA	90005
Mr	Peters	Los Angeles	CA	90013
Mr	Randolph	Los Angeles	CA	90018
Mrs	Smith	Los Angeles	CA	90011
Comp	TOOLware	Los Angeles	CA	90002

## 2.2 Selecting Particular Columns

Columns are selected from a base table simply by specifying their names after the keyword SELECT and separating them by commas. Blanks may be used, but are not needed.

```
SELECT name, firstname  
FROM customer
```

NAME	FIRSTNAME
Porter	Jenny
DATASOFT	?
Randolph	Martin
Smith	Sally
Brown	Peter
Jackson	Michael
Howe	George
Miller	Frank
Peters	Joseph
Baker	Susan
Jenkins	Anthony
Adams	Thomas
Griffith	Mark
TOOLware	?
Brown	Rose

## 2.3 Ordering Columns

The selected columns are arranged in the order of the column names, as they are specified in the SELECT statement and not as they are stored in the base table.

```
SELECT firstname, name  
FROM customer
```

FIRSTNAME	NAME
Jenny	Porter
?	DATASOFT
Martin	Randolph
Sally	Smith
Peter	Brown
Michael	Jackson
George	Howe
Frank	Miller
Joseph	Peters
Susan	Baker
Anthony	Jenkins
Thomas	Adams
Mark	Griffith
?	TOOLware
Rose	Brown

An '\*' can be used instead of the list of column names. The user receives all the table columns in the order defined in the database.

```
SELECT *  
FROM customer
```

produces the table 'customer' with all its columns and rows as the result.



## 2.4 Selecting Particular Rows

In addition to restricting the number of columns in a base table, it is also possible to restrict the number of rows. Rows are selected by formulating a WHERE condition.

Selection of rows with the city 'New York':

```
SELECT title, firstname, name, city
FROM customer
WHERE city = 'New York'
```

TITLE	FIRSTNAME	NAME	CITY
Mrs	Jenny	Porter	New York
Mr	George	Howe	New York
Mr	Mark	Griffith	New York

Selection of rows with an account of 0.00:

```
SELECT name, city, account
FROM customer
WHERE account = 0.00
```

NAME	CITY	ACCOUNT
Randolph	Los Angeles	0.00
Smith	Los Angeles	0.00
Brown	Hollywood	0.00
Jackson	Washington	0.00
Miller	Chicago	0.00
Jenkins	Los Angeles	0.00
Griffith	New York	0.00

Now those rows are to be selected which have no value in a specified column.

"No value" is *not* indicated by the value "0" or " ", but by NULL.

```
SELECT title, firstname, name
FROM customer
WHERE firstname IS NULL
```

TITLE	FIRSTNAME	NAME
Comp Comp	? ?	DATASOFT TOOLware

```
SELECT name, city
FROM customer
WHERE account IS NULL
```

```
***ERROR 100 ROW NOT FOUND
```

If only the first five columns are to be output, and provided with numbers, the statement is as follows:

```
SELECT ROWNO, cno, title, firstname, name
FROM customer
WHERE ROWNO <= 5
```

Then the result is:

ROWNO	CNO	TITLE	FIRSTNAME	NAME
1	3000	Mrs	Jenny	Porter
2	3100	Comp	?	DATASOFT
3	3200	Mr	Martin	Randolph
4	3300	Mrs	Sally	Smith
5	3400	Mr	Peter	Brown

## 2.6 Ordering Rows

ORDER BY specifies the order in which the rows are to be output.

In the first example, all rows are alphabetically ordered according to 'name'.

```
SELECT name, firstname, city, account
FROM customer
ORDER BY name
```

NAME	FIRSTNAME	CITY	ACCOUNT
Adams	Thomas	Los Angeles	-416.88
Baker	Susan	Los Angeles	-4167.79
Brown	Peter	Hollywood	0.00
Brown	Rose	Hollywood	440.00
DATASOFT	?	Dallas	4813.50
Griffith	Mark	New York	0.00
Howe	George	New York	-315.40
Jackson	Michael	Washington	0.00
Jenkins	Anthony	Los Angeles	0.00
Miller	Frank	Chicago	0.00
Peters	Joseph	Los Angeles	650.00
Porter	Jenny	New York	100.00
Randolph	Martin	Los Angeles	0.00
Smith	Sally	Los Angeles	0.00
TOOLware	?	Los Angeles	3770.50

In the next example, the rows are arranged in numerically descending (DESC) order. If no order is specified for a sort column, ascending order is always the implicit sorting default. Ascending can also be explicitly specified with ASC.

```
SELECT name, firstname, city, account
FROM customer
ORDER BY account DESC
```

NAME	FIRSTNAME	CITY	ACCOUNT
DATASOFT	?	Dallas	4813.50
TOOLware	?	Los Angeles	3770.50
Peters	Joseph	Los Angeles	650.00
Brown	Rose	Hollywood	440.00
Porter	Jenny	New York	100.00
Randolph	Martin	Los Angeles	0.00
Smith	Sally	Los Angeles	0.00
Brown	Peter	Hollywood	0.00
Jackson	Michael	Washington	0.00
Miller	Frank	Chicago	0.00
Jenkins	Anthony	Los Angeles	0.00
Griffith	Mark	New York	0.00
Howe	George	New York	-315.40
Adams	Thomas	Los Angeles	-416.88
Baker	Susan	Los Angeles	-4167.79

A sort column need not be an output column as well.

The position number may be specified in the output list instead of the sort column name:

```
SELECT name, firstname, city, account
FROM customer
ORDER BY 4 DESC
```

The order for ASCII code is:

1. Blanks
2. Special characters (%,&,+,-,\*,/,...)
3. Numbers
4. Special characters (:,;,,<,>,...)
5. Capital letters
6. Small letters
7. Null value

The order for EBCDIC code is:

1. Blanks
2. Special characters (+,&,\*,,,-,/,%,;,...)
3. Small letters
4. Capital letters
5. Numbers
6. Null value

The type of coding is established while defining the table (see below).

To obtain a useful handling and sorting of umlauts and special letters occurring in other languages, ADABAS uses so-called 'MAPCHAR SETs'. A DEFAULTMAP for the conversion of country-specific letters is created during the installation of the database.

The database administrator can create MAPCHAR SETs of his own. When doing so, he defines the way in which special letters have to be mapped to other letters, thus influencing the sort sequence.

For example, if 'ü' (in ASCII representation X'FC') is to be sorted as 'ue', the following assignment must be made:

FC ... ue

If 'ü' is to be sorted as 'u', then the line looks like this:

FC ... u

To obtain a particular sort sequence for a result table, the function MAPCHAR must be used. If the sort sequence is to deviate from the DEFAULTMAP, a MAPCHAR SET defined by the administrator must be specified as argument of the function.

The statement

```
SELECT name, firstname, city, account, MAPCHAR(name) lname
FROM customer
ORDER BY lname
```

produces the desired sort sequence.

## 2.7 Ordering According to More than One Column

To perform a sorting according to more than one criterion, all sort column names must be entered in the order of their importance. Each column name can be qualified by ASC or DESC. These qualifications can also be mixed.

In the next two examples, the sorting is done according to two criteria, the order - and therefore the importance - of which is interchanged.

In the first case, the main criterion is the city, the secondary criterion the account.

```
SELECT name, city, account
      FROM customer
      ORDER BY city, account DESC
```

NAME	CITY	ACCOUNT
Miller	Chicago	0.00
DATASOFT	Dallas	4813.50
Brown	Hollywood	440.00
Brown	Hollywood	0.00
TOOLware	Los Angeles	3770.50
Peters	Los Angeles	650.00
Randolph	Los Angeles	0.00
Smith	Los Angeles	0.00
Jenkins	Los Angeles	0.00
Adams	Los Angeles	-416.88
Baker	Los Angeles	-4167.79
Porter	New York	100.00
Griffith	New York	0.00
Howe	New York	-315.40
Jackson	Washington	0.00

In the following example, the main criterion is the account, the secondary criterion the city.

```
SELECT name, city, account
FROM customer
ORDER BY account DESC, city
```

NAME	CITY	ACCOUNT
DATASOFT	Dallas	4813.50
TOOLware	Los Angeles	3770.50
Peters	Los Angeles	650.00
Brown	Hollywood	440.00
Porter	New York	100.00
Miller	Chicago	0.00
Brown	Hollywood	0.00
Randolph	Los Angeles	0.00
Smith	Los Angeles	0.00
Jenkins	Los Angeles	0.00
Griffith	New York	0.00
Jackson	Washington	0.00
Howe	New York	-315.40
Adams	Los Angeles	-416.88
Baker	Los Angeles	-4167.79



## 2.8 Eliminating Duplicate Rows

All customer cities are to be retrieved. First all cities are output as often as they are stored.

```
SELECT city, state
FROM customer
ORDER BY state
```

CITY	STATE
Los Angeles	CA
Los Angeles	CA
Hollywood	CA
Los Angeles	CA
Los Angeles	CA
Los Angeles	CA
Los Angeles	CA
Los Angeles	CA
Hollywood	CA
Washington	DC
Chicago	IL
New York	NY
New York	NY
New York	NY
Dallas	TX

This redundancy can be avoided by the usage of DISTINCT.

```
SELECT DISTINCT city, state
      FROM customer
      ORDER BY city
```

CITY	STATE
Chicago	IL
Dallas	TX
Hollywood	CA
Los Angeles	CA
New York	NY
Washington	DC

## 2.9 Keeping Result Sets

If a user wants to reuse a result table in other queries, he must name it and specify FOR REUSE for it. Then the table can be addressed within a transaction by this name at any time if it has not been explicitly deleted or another SELECT specifying a result table with the same name has not been performed. When leaving the session, the result table is automatically deleted.

```
SELECT report1 (title, firstname, name)
      FROM customer
      WHERE city = 'New York'
      FOR REUSE
```

TITLE	FIRSTNAME	NAME
Mrs	Jenny	Porter
Mr	George	Howe
Mr	Mark	Griffith

```
SELECT title, name
      FROM report1
```

TITLE	NAME
Mrs	Porter
Mr	Howe
Mr	Griffith

## 2.10 Introducing New Result Column Names

In the examples shown so far, the column names of the base table were used for the column names of the result table.

It is possible to give result columns new names.

```
SELECT title mrmrs, firstname christianname
       name surname, city address
FROM customer
WHERE city = 'New York'
```

MRMRS	CHRISTIANNAME	SURNAME	ADDRESS
Mrs	Jenny	Porter	New York
Mr	George	Howe	New York
Mr	Mark	Griffith	New York

Of course, these two forms of renaming result sets and result columns can be combined in a statement.

```
SELECT report2 (name customer, city dispatch, account)
FROM customer
WHERE account = 0.0
FOR REUSE
```

CUSTOMER	DISPATCH	ACCOUNT
Randolph	Los Angeles	0.00
Smith	Los Angeles	0.00
Brown	Hollywood	0.00
Jackson	Washington	0.00
Miller	Chicago	0.00
Jenkins	Los Angeles	0.00
Griffith	New York	0.00

## 3Conditional Selection

The next pages show various conditions that can be written after WHERE.

Apart from EQUAL TO (written '=') there are:

LESS THAN	<
LESS THAN OR EQUAL TO	<=

GREATER THAN	>
GREATER THAN OR EQUAL TO	>=

NOT EQUAL TO	<>
--------------	----

For several conditions	AND, OR
For negative conditions	NOT

For values in a range	BETWEEN x AND y
For values in a set	IN (x,y,z)

For comparing partial values	LIKE '%abc%', LIKE '*abc*' LIKE '_a_', LIKE '?a?' LIKE '*@?' ESCAPE '@'
------------------------------	---

For comparing similarly sounding values	SOUNDS
--	--------

Query for the NULL value	IS NULL
--------------------------	---------

Query for a Boolean value	IS TRUE IS FALSE
---------------------------	---------------------

Selection without condition:

```
SELECT city, name, firstname
FROM customer
```

CITY	NAME	FIRSTNAME
New York	Porter	Jenny
Dallas	DATASOFT	?
Los Angeles	Randolph	Martin
Los Angeles	Smith	Sally
Hollywood	Brown	Peter
Washington	Jackson	Michael
New York	Howe	George
Chicago	Miller	Frank
Los Angeles	Peters	Joseph
Los Angeles	Baker	Susan
Los Angeles	Jenkins	Anthony
Los Angeles	Adams	Thomas
New York	Griffith	Mark
Los Angeles	TOOLware	?
Hollywood	Brown	Rose

Selection with condition:

```
SELECT city, name, firstname
FROM customer
WHERE city = 'Los Angeles'
```

CITY	NAME	FIRSTNAME
Los Angeles	Randolph	Martin
Los Angeles	Smith	Sally
Los Angeles	Peters	Joseph
Los Angeles	Baker	Susan
Los Angeles	Jenkins	Anthony
Los Angeles	Adams	Thomas
Los Angeles	TOOLware	?

### 3.1 Comparison Operations

Comparison conditions are formulated using the following symbols:

=	equal to
<>	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Which customer has an empty or a positive account?

```
SELECT title, name, account
FROM customer
WHERE account >= 0
```

TITLE	NAME	ACCOUNT
Mrs	Porter	100.00
Comp	DATASOFT	4813.50
Mr	Randolph	0.00
Mrs	Smith	0.00
Mr	Brown	0.00
Mr	Jackson	0.00
Mr	Miller	0.00
Mr	Peters	650.00
Mr	Jenkins	0.00
Mr	Griffith	0.00
Comp	TOOLware	3770.50
Mrs	Brown	440.00

Which customer has a positive account?

```
SELECT title, name, account
      FROM customer
     WHERE account > 0
```

TITLE	NAME	ACCOUNT
Mrs	Porter	100.00
Comp	DATASOFT	4813.50
Mr	Peters	650.00
Comp	TOOLware	3770.50
Mrs	Brown	440.00

Which customers are companies?

```
SELECT title, name
      FROM customer
     WHERE title = 'Comp'
```

TITLE	NAME
Comp	DATASOFT
Comp	TOOLware



Selection of all customers who, in alphabetical order, follow 'Peters':

```
SELECT firstname, name, city
  FROM customer
 WHERE name > 'Peters'
```

FIRSTNAME	NAME	CITY
Jenny	Porter	New York
Martin	Randolph	Los Angeles
Sally	Smith	Los Angeles
?	TOOLware	Los Angeles

### 3.2 Multiple Conditions: AND, OR

It is possible to retrieve rows which satisfy several conditions combined by AND or OR.

The first example shows customers who live in New York or have a positive account.

```
SELECT firstname, name, city, account
FROM customer
WHERE city = 'New York' OR account > 0
```

FIRSTNAME	NAME	CITY	ACCOUNT
Jenny	Porter	New York	100.00
?	DATASOFT	Dallas	4813.50
George	Howe	New York	-315.40
Joseph	Peters	Los Angeles	650.00
Mark	Griffith	New York	0.00
?	TOOLware	Los Angeles	3770.50
Rose	Brown	Hollywood	440.00

The second example only shows those customers from New York who have a positive account.

```
SELECT firstname, name, city, account
FROM customer
WHERE city = 'New York' AND account > 0
```

FIRSTNAME	NAME	CITY	ACCOUNT
Jenny	Porter	New York	100.00

### Parentheses with AND and OR

Whenever AND and OR are used together, it is recommended to use parentheses in order to clarify the instruction.

Example of a WHERE condition:

```
title = 'Comp' AND state = 'TX' OR account > 0.0
```

If the parentheses are applied in this way:

```
(title = 'Comp' AND state = 'TX') OR (account > 0.0)
```

the result consists of all customers who are Texan companies, irrespective of their accounts, plus all customers who have positive accounts. The result of this query is identical to that without parentheses, because AND has a higher precedence than OR.

If the parentheses are set differently:

```
(title = 'Comp') AND (state = 'TX' OR account > 0.0)
```

then only 'company' customers are found who are either located in Texas or who have a positive account. Of course, customers who satisfy both conditions - this is, Texan companies with a positive account - are output as well.

Nesting of parentheses is possible. The content of the innermost parentheses is evaluated first.

### 3.3 Ranges of Values: BETWEEN x AND y

All rows can be searched which have a value within a specified range.

```
account BETWEEN -420 AND 0
```

means that all accounts having a value between -420 and 0 (both values included) will be found. This condition can also be written in the following way:

```
account >= -420 AND account <= 0.
```

```
SELECT title, name, city, account
FROM customer
WHERE account BETWEEN -420 AND 0
```

TITLE	NAME	CITY	ACCOUNT
Mr	Randolph	Los Angeles	0.00
Mrs	Smith	Los Angeles	0.00
Mr	Brown	Hollywood	0.00
Mr	Jackson	Washington	0.00
Mr	Howe	New York	-315.40
Mr	Miller	Chicago	0.00
Mr	Jenkins	Los Angeles	0.00
Mr	Adams	Los Angeles	-416.88
Mr	Griffith	New York	0.00

Selection of all customers whose name begins with the letter 'B':

```
SELECT title, name
      FROM customer
     WHERE name BETWEEN 'Ba' AND 'Bz'
```

TITLE	NAME
Mr	Brown
Mrs	Baker
Mrs	Brown

This query can be formulated more 'elegantly' using LIKE. Section 3.5 contains a description of how this can be done.

Selection of all customers who live in Los Angeles:

```
SELECT title, name, city, state, zip
      FROM customer
     WHERE zip BETWEEN 9000 AND 90024
```

TITLE	NAME	CITY	STATE
Mr	Randolph	Los Angeles	CA
Mrs	Smith	Los Angeles	CA
Mr	Peters	Los Angeles	CA
Mrs	Baker	Los Angeles	CA
Mr	Jenkins	Los Angeles	CA
Mr	Adams	Los Angeles	CA
Comp	TOOLware	Los Angeles	CA

### 3.4 Values in a Set: IN (x,y,z)

If the number of possible values is small, so that it is easy to list them, they can be combined to form a set using parentheses and placing the IN operator before them.

The order of the values within the parentheses is not relevant.

Selection of all customers who are not companies:

```
SELECT title, firstname, name, city
FROM customer
WHERE title IN ('Mr','Mrs')
```

TITLE	FIRSTNAME	NAME	CITY
Mrs	Jenny	Porter	New York
Mr	Martin	Randolph	Los Angeles
Mrs	Sally	Smith	Los Angeles
Mr	Peter	Brown	Hollywood
Mr	Michael	Jackson	Washington
Mr	George	Howe	New York
Mr	Frank	Miller	Chicago
Mr	Joseph	Peters	Los Angeles
Mrs	Susan	Baker	Los Angeles
Mr	Anthony	Jenkins	Los Angeles
Mr	Thomas	Adams	Los Angeles
Mr	Mark	Griffith	New York
Mrs	Rose	Brown	Hollywood

### 3.5 Searching For Character Strings

If not all characters are known in a column, incomplete search values can be used. It is possible to search for character strings within a column. This is not supported for numeric columns.

- The number of characters is unknown: LIKE '%abc%'

All values that contain the character string 'abc' are searched. This string can be preceded or followed by any number of characters or no character at all.

Find all customers with 'SOFT' at the end of their names:

```
SELECT name, city
FROM customer
WHERE name LIKE '%SOFT'
```

NAME	CITY
DATASOFT	Dallas

The character '\*' can be used instead of '%'.

- A fixed number of characters is known: LIKE '\_c\_'

If the number of unknown characters is fixed and known, the positions in question can be exactly determined.

An alternative notation for '\_' is '?'.

Find all customers whose names consist of six letters and begin with 'P':

```
SELECT name, city
FROM customer
WHERE name LIKE 'P?????'
```

NAME	CITY
Porter	New York
Peters	Los Angeles

Find all customers whose first names have any lengths and begin with 'M':

```
SELECT firstname, name, city
FROM customer
WHERE firstname LIKE 'M%'
```

FIRSTNAME	NAME	CITY
Martin	Randolph	Los Angeles
Michael	Jackson	Washington
Mark	Griffith	New York



Which customer names have an 'o' anywhere after the first letter?

```
SELECT name, city
FROM customer
WHERE name LIKE '_%o%'
```

NAME	CITY
Porter	New York
Randolph	Los Angeles
Brown	Hollywood
Jackson	Washington
Howe	New York
Brown	Hollywood

If one of the special characters \*, %, ?, \_ is to be searched within the table rows, it must be masked using an ESCAPE character. This character can be chosen freely.

Find all customers whose names contain an '\_'. In this example, the @ sign is used as the ESCAPE character.

```
SELECT name, city
FROM customer
WHERE name LIKE '%@_%' ESCAPE '@'
```

### 3.6 Negative Conditions: NOT

To obtain the opposite of a condition, NOT must be placed before the relevant expression. The negation refers to the following expression. If a compound expression is to be negated, it must be enclosed in parentheses.

According to the part to be negated, the expression NOT x AND y OR z can be parenthesized in the following ways. Only the first two expressions are logically equivalent, i.e., have the same result.

	NOT x AND y OR z
	NOT (x) AND y OR z
or	NOT (x AND y) OR z
or	NOT (x AND y OR z),

```
SELECT name, city, state, zip
FROM customer
WHERE NOT (city = 'Dallas' OR
          city = 'New York')
```

NAME	CITY	STATE	ZIP
Randolph	Los Angeles	CA	90018
Smith	Los Angeles	CA	90011
Brown	Hollywood	CA	90029
Jackson	Washington	DC	20037
Miller	Chicago	IL	60601
Peters	Los Angeles	CA	90013
Baker	Los Angeles	CA	90008
Jenkins	Los Angeles	CA	90005
Adams	Los Angeles	CA	90014
TOOLware	Los Angeles	CA	90002
Brown	Hollywood	CA	90025

### 3.7 NOT with NULL, LIKE, IN, BETWEEN

NULL, LIKE, IN, and BETWEEN are operators which can be preceded by NOT. In the case of NULL, the word 'IS' is needed. The condition

```
WHERE NOT (firstname IS NULL)
```

can also be written as

```
WHERE firstname IS NOT NULL.
```

Find all customers who have a first name, i.e., are not companies:

```
SELECT firstname, name, city  
FROM customer  
WHERE firstname IS NOT NULL
```

FIRSTNAME	NAME	CITY
Jenny	Porter	New York
Martin	Randolph	Los Angeles
Sally	Smith	Los Angeles
Peter	Brown	Hollywood
Michael	Jackson	Washington
George	Howe	New York
Frank	Miller	Chicago
Joseph	Peters	Los Angeles
Susan	Baker	Los Angeles
Anthony	Jenkins	Los Angeles
Thomas	Adams	Los Angeles
Mark	Griffith	New York
Rose	Brown	Hollywood

Find the customers who are not a company:

```
SELECT name, city
FROM customer
WHERE title NOT LIKE 'Co%'
```

NAME	CITY	STATE	ZIP
Randolph	Los Angeles	CA	90018
Smith	Los Angeles	CA	90011
Brown	Hollywood	CA	90029
Jackson	Washington	DC	20037
Miller	Chicago	IL	60601
Peters	Los Angeles	CA	90013
Baker	Los Angeles	CA	90008
Jenkins	Los Angeles	CA	90005
Adams	Los Angeles	CA	90014
TOOLware	Los Angeles	CA	90002
Brown	Hollywood	CA	90025

Find the customers who do not live in Dallas or New York:

```
SELECT name, city, state, zip
FROM customer
WHERE city NOT IN ('Dallas','New York')
```

NAME	CITY	STATE	ZIP
Randolph	Los Angeles	CA	90018
Smith	Los Angeles	CA	90011
Brown	Hollywood	CA	90029
Jackson	Washington	DC	20037
Miller	Chicago	IL	60601
Peters	Los Angeles	CA	90013
Baker	Los Angeles	CA	90008
Jenkins	Los Angeles	CA	90005
Adams	Los Angeles	CA	90014
TOOLware	Los Angeles	CA	90002
Brown	Hollywood	CA	90025

Find the customers who have either a positive account or a considerable negative account:

```
SELECT title, name, city, account
FROM customer
WHERE account NOT BETWEEN -10 AND 0
```

TITLE	NAME	CITY	ACCOUNT
Mrs	Porter	New York	100.00
Comp	DATASOFT	Dallas	4813.50
Mr	Howe	New York	-315.40
Mr	Peters	Los Angeles	650.00
Mrs	Baker	Los Angeles	-4167.79
Mr	Adams	Los Angeles	-416.88
Comp	TOOLware	Los Angeles	3770.50
Mrs	Brown	Hollywood	440.00

The preceding example can also be formulated differently:

```
SELECT title, name, city, account
FROM customer
WHERE NOT (account >= -10 AND ACCOUNT <= 0)
```

or else:

```
SELECT title, name, city, account
FROM customer
WHERE account < -10 OR account > 0
```

### 3.8 Grouping Values: GROUP BY

Example:

```
SELECT city, FIXED (AVG(account),7,2)
FROM customer
GROUP BY city
ORDER BY city
```

CITY	EXPRESSION1
Chicago	0.00
Dallas	4813.50
Hollywood	220.00
Los Angeles	-23.45
New York	-71.80
Washington	0.00

The query determines the average account for each city (AVG). As the second column of the result table has no longer a predefined name, the system gives the default heading EXPRESSION1 to it. GROUP BY arranges the table in groups of rows with the same city name and produces one result row for each group. ORDER BY should be used for sorted output of the group results. GROUP BY collects the results by groups; but it does not necessarily sort them.

One of the functions that are applied to a whole column of a temporary result table must be written before all the columns, except 'city'. (These functions are here also called 'set functions'.) 'city' does not need a function specification, because each element of the group has the same value for 'city'.

AVG is a function that is applied to a whole column, like MIN, MAX, COUNT, SUM, STDDEV, and VARIANCE. The next section contains a more detailed explanation of these functions.

GROUP BY usually produces a group for each different value of the column to be grouped.

Grouping can be done according to several columns. To begin with, groups are formed according to the first criterion. Then each of these groups is arranged according to the next criterion, and so on.

If GROUP BY is used, it must follow FROM and WHERE and precede ORDER BY.

Show the minimum, the average, and the maximum account of all customers for each city:

```
SELECT city, MIN(account) min_account,
        FIXED (AVG(account),7,2) avg_account,
        MAX(account) max_account
FROM customer
GROUP BY city
```

CITY	MIN_ACCOUNT	AVG_ACCOUNT	MAX_ACCOUNT
Chicago	0.00	0.00	0.00
Dallas	4813.50	4813.50	4813.50
Hollywood	0.00	220.00	440.00
Los Angeles	-4167.79	-23.45	3770.50
New York	-315.40	-71.80	100.00
Washington	0.00	0.00	0.00

New headings were defined for the calculated result columns.



Show the minimum, the average, and the maximum account of all customers for each city, except New York.

```
SELECT city, MIN(account) min_account,
        FIXED (AVG(account),7,2) avg_account,
        MAX(account) max_account
FROM customer
WHERE city <> 'New York'
GROUP BY city
```

CITY	MIN ACCOUNT	AVG ACCOUNT	MAX ACCOUNT
Chicago	0.00	0.00	0.00
Dallas	4813.50	4813.50	4813.50
Hollywood	0.00	220.00	440.00
Los Angeles	-4167.79	-23.45	3770.50
Washington	0.00	0.00	0.00

Show the number of customers and the total of their accounts for each city:

```
SELECT city, COUNT(*) number,
        FIXED (AVG(account),7,2) avg_account,
        SUM(account) sum_account
FROM customer
GROUP BY city
```

CITY	NUMBER	AVG ACCOUNT	SUM ACCOUNT
Chicago	1	0.00	0.00
Dallas	1	4813.50	4813.50
Hollywood	2	220.00	440.00
Los Angeles	7	-23.45	-164.17
New York	3	-71.80	-215.40
Washington	1	0.00	0.00

### 3.9 Groups with Conditions: HAVING

Example:

```
SELECT city, COUNT(*) number,  
       FIXED (AVG(account),7,2) avg_account,  
       SUM(account) sum_account  
FROM customer  
GROUP BY city  
HAVING COUNT(*) > 1
```

CITY	NUMBER	AVG_ACCOUNT	SUM_ACCOUNT
Hollywood	2	220.00	440.00
Los Angeles	7	-23.45	-164.17
New York	3	-71.80	-215.40

Compare the requesting command with the previous example. The line 'HAVING COUNT(\*) > 1' eliminates all cities with only one customer.

The question could be asked, when HAVING is used instead of WHERE. Actually, their usage is much alike. A condition with WHERE excludes data from the table to be searched through, whereas HAVING is normally used after a function that is applied to a whole column.

Columns in the output list for which none of these functions has been issued must be grouped using GROUP BY. If a condition after HAVING is not satisfied, *groups* of values are excluded from the output. If one of the functions described above is applied to *all* columns of the output list, GROUP BY is not needed, and the condition is only checked for the only output row.

(Compare the following examples with those of section 3.8.)

Show the minimum, the average, and the maximum account of all customers with an average account  $\geq 0.00$  for all cities, except New York.

```
SELECT city, MIN(account) min_account,
        FIXED (AVG(account),7,2) avg_account,
        MAX(account) max_account
FROM customer
WHERE city <> 'New York'
GROUP BY city
HAVING AVG(account) >= 0.00
```

CITY	MIN_ACCOUNT	AVG_ACCOUNT	MAX_ACCOUNT
Chicago	0.00	0.00	0.00
Dallas	4813.50	4813.50	4813.50
Hollywood	0.00	220.00	440.00
Washington	0.00	0.00	0.00

For each city with more than two customers, show the number of customers living there and the average account:

```
SELECT city, COUNT(*) number, FIXED (AVG(account),7,2) avg_account
FROM customer
GROUP BY city
HAVING COUNT(*) > 2
```

CITY	NUMBER	AVG_ACCOUNT
Los Angeles	7	-23.45
New York	3	-71.80

Show the summed up account for each city with a total account of -100.00 at the most and two customers at least:

```
SELECT city, SUM(account) sum_account
FROM customer
GROUP BY city
HAVING COUNT(*) >= 2
AND SUM(account) <= -100.00
```

CITY	SUM_ACCOUNT
Los Angeles	-164.17
New York	-215.40

## 4 Virtual Result Columns

### 4.1 Expanding the Sample

The examples of SQL statements used so far were based on the table 'customer'. They dealt with restrictions for this table and the formatting of the output (reordering, headings, and the like).

This sample is expanded in the following by three tables, 'hotel', 'room', and 'reservation', in order to represent more complex calculations and interrelations of several tables. The way of putting the questions is simplified. The questions themselves could come from the activities of a travel agency.

The table 'hotel' describes various hotels and their addresses. The hotels are identified by a unique number.

HNO	NAME	CITY	STATE	ZIP	ADDRESS
10	Congress	Detroit	MI	48226	155 Beechwood Str.
20	Long Island	Cincinnati	OH	45211	1499 Grove Street
30	Regency	Portland	OR	97213	477 17th Avenue
40	Eight Avenue	Chicago	IL	60601	112 8th Avenue
50	Lake Michigan	Chicago	IL	60615	354 OAK Terrace
60	Airport	New Orleans	LA	70112	650 C Parkway
70	Empire State	New York	NY	10019	65 Yellowstone Dr.
80	Midtown	Chicago	IL	60607	12 Barnard Street
90	Long Beach	Long Beach	CA	90804	200 Yellowstone Dr
100	Dallas	Dallas	TX	75225	1980 34th Str.
110	Atlantic	New York	NY	10570	111 78th Str.
120	Sunshine	Los Angeles	CA	90018	35 Broadway
130	Star	Hollywood	CA	90030	13 Beechwood Place
140	River Boat	Washington	DC	20019	788 MAIN STREET
150	Indian Horse	Santa Clara	CA	95054	16 MAIN STREET

The table 'room' contains both the particular prices for rooms of different sizes and the total number of rooms available for this category. The association of the rooms with the hotels is ensured by the hotel number.

HNO	ROOMTYPE	MAX_FREE	PRICE
10	single	20	135.00
10	double	45	200.00
30	single	12	45.00
30	double	15	80.00
20	single	10	70.00
20	double	13	100.00
70	single	4	115.00
70	double	11	180.00
80	single	15	90.00
80	double	19	150.00
80	suite	5	400.00
40	single	20	85.00
40	double	35	140.00
50	single	50	105.00
50	double	230	180.00
50	suite	12	500.00
60	single	10	120.00
60	double	39	200.00
60	suite	20	500.00
90	single	45	90.00
90	double	145	150.00
90	suite	60	300.00
100	single	11	60.00
100	double	24	100.00
110	single	2	70.00
110	double	10	130.00
120	single	34	80.00
120	double	78	140.00
120	suite	55	350.00
150	single	44	100.00
150	double	115	190.00
150	suite	6	450.00
130	single	89	160.00
130	double	300	270.00
130	suite	100	700.00
140	single	10	125.00
140	double	9	200.00
140	suite	78	600.00

In the table 'reservation', at last, it is recorded which customer made a reservation in which hotel for which category of room. Thus a logical relation between the tables 'customer', 'hotel', and 'room' is established. The entries are provided with unique numbers for identification purposes.

RNO	CNO	HNO	ROOMTYPE	ARRIVAL	DEPARTURE
100	3000	80	single	11/13/1994	11/15/1994
110	3000	100	double	12/24/1994	01/06/1995
120	3200	50	suite	11/14/1994	11/18/1994
130	3900	110	single	02/01/1995	02/03/1995
140	4300	80	double	04/12/1994	04/30/1994
150	3600	70	double	03/14/1995	03/24/1995
160	4100	70	single	04/12/1994	04/15/1994
170	4400	150	suite	09/01/1994	09/03/1994
180	3100	120	double	12/23/1994	01/08/1995
190	4300	140	double	11/14/1994	11/17/1994

The complete definition of the tables is included in section 12.

## 4.2 Arithmetic within the Result List

An SQL query cannot only find data which already exists in a table, but also values which may be calculated from this data.

First the daily prices of the single rooms of all hotels are to be selected:

```
SELECT hno, roomtype, price
FROM room
WHERE roomtype = 'single'
```

HNO	ROOMTYPE	PRICE
10	single	135.00
30	single	45.00
20	single	70.00
70	single	115.00
80	single	90.00
40	single	85.00
50	single	105.00
60	single	120.00
90	single	90.00
100	single	60.00
110	single	70.00
120	single	80.00
150	single	100.00
130	single	160.00
140	single	125.00



To obtain the weekly price of a hotel room, the statement can be formulated in the following way:

```
SELECT hno, roomtype, price*7 price_of_week  
FROM room  
WHERE roomtype = 'single'
```

HNO	ROOMTYPE	PRICE_OF_WEEK
10	single	955.00
30	single	315.00
20	single	490.00
70	single	805.00
80	single	630.00
40	single	595.00
50	single	735.00
60	single	840.00
90	single	630.00
100	single	420.00
110	single	490.00
120	single	560.00
150	single	700.00
130	single	1120.00
140	single	875.00

'price\*7' is an expression. The weekly price is calculated as the result of the multiplication of 'price' by 7.

### 4.3 Expressions with +, -, \*, / , DIV, MOD

Expressions can be formed

for the operations	using	the symbols
--------------------	-------	-------------

addition		+
subtraction		-
multiplication		*
division		/

Column names (speed \* time), constants (speed \* 1.01), and functions related to a whole column, such as (AVG (account - 500)), can be used. An expression can be a numeric constant or a character value.

Furthermore, the following functions are provided:

A DIV B	for the integer division of A by B
---------	------------------------------------

A MOD B	for the remainder of an integer division of A by B
---------	--

## 4.4 Arithmetic Operations

Additional arithmetic operators for numeric values A, B are:

TRUNC (A)	for truncating the decimal places of A
TRUNC (A,n)	for truncating the number A after n digits to the right of the decimal point
TRUNC (A,-n)	for setting n digits to the left of the decimal point of the number A to 0
ROUND (A)	for rounding the decimal places
ROUND (A,n)	for rounding the nth digit to the right of the decimal point
ROUND (A,-n)	for rounding n digits to the left of the decimal point
NOROUND (A)	for preventing the rounding of values performed to adapt the values to a specific data type
CEIL (A)	for forming the smallest integer value greater than or equal to A
FLOOR (A)	for forming the greatest integer value less than or equal to A
SIGN (A)	for giving information about the sign of A
ABS (A)	for the unsigned (absolute) value of A
FIXED (a,p,q)	for specifying the number a in a format of the data type FIXED (p,q) with rounding
POWER (A,n)	for forming the nth power of A
SQRT (A)	for calculating the square root of A

EXP (A)	for forming the power from the base of e (2.71828183) and the exponent A ("e to the power A")
LN (A)	for forming the natural logarithm of A
LOG (A,B)	for forming the logarithm of B to the base of A
PI	for displaying the value of $\pi$

## 4.5 Trigonometric Functions

The following trigonometric functions producing a numeric value as the result are provided:

COS (A)	cosine of the number A
SIN (A)	sine of the number A
TAN (A)	tangent of the number A
COT (A)	cotangent of the number A
COSH (A)	hyperbolic cosine of the number A
SINH (A)	hyperbolic sine of the number A
TANH (A)	hyperbolic tangent of the number A
ACOS (A)	arc cosine of the number A
ASIN (A)	arc sine of the number A
ATAN (A)	arc tangent of the number A
ATAN2 (A,B)	forms the arc tangent of the value A/B under certain circumstances
RADIANS (A)	the angle in radians of the number A
DEGREES (A)	measure of degree of the number A

## 4.6 Date and Time Calculations

To facilitate the handling of date and time calculations, several functions are available which compute with values of these types.

A reservation date increased by two days gives:

```
SELECT arrival, ADDDATE (arrival,2) arrival2, rno
FROM   reservation
WHERE  rno = 130
```

ARRIVAL	ARRIVAL2	RNO
02/01/1995	02/03/1995	130

The number of reservation days between arrival and departure gives:

```
SELECT arrival, departure, DATEDIFF (arrival, departure)
      difference, rno
FROM reservation
WHERE rno = 130
```

ARRIVAL	DEPARTURE	DIFFERENCE	RNO
02/01/1995	02/03/1995	2	130

Additional date functions are:

SUBDATE	computes a past date
DAYOFWEEK	indicates the day of week (first day: Monday)
DAYOFMONTH	indicates the number of the day of month
DAYOFYEAR	indicates the number of the day of year
WEEKOFYEAR	indicates the number of the week of year for the specified day
YEAR, MONTH, DAY	extract the year, month or day from a date or timestamp value
MAKEDATE	forms a date value from a year and a day
DAYNAME	displays the day of week as a character string
MONTHNAME	displays the name of month as a character string

The corresponding time functions are:

ADDTIME  
SUBTIME  
TIMEDIFF

HOURL  
MINUTE  
SECOND  
MICROSECOND

MAKETIME	forms a time value out of three significant numbers
TIMESTAMP	forms a timestamp value consisting of a date, a time value and 0 micro seconds

DATE	forms a date value
TIME	forms a time value

Different date formats are available for the processing of date values. The keywords ISO, USA, EUR, JIS or INTERNAL can be used to determine that the date is to be represented either according to the ISO standard or in US American, European, Japanese or the database internal format (see the ADABAS Reference Manual).

#### **4.7 Evaluation Sequence**

The operations in an expression are evaluated in the following sequence:

1. Plus or minus before a single value
2. Multiplication and division of two values
3. Addition and subtraction of two values

Operations of the same precedence are evaluated from left to right.

Parentheses can be used to determine the evaluation sequence. For example, the following two expressions are equivalent:

$A * -B / C + D / E$       and       $((A * (-B)) / C) + (D / E)$

Data type conversions (FIXED - FLOAT), conflicts of ranges of values and arithmetic overflows are controlled by the system.



## 4.8 Arithmetic in a Condition

Column names can also be used within conditional expressions.

```
SELECT hno, roomtype, price
FROM room
WHERE price*7 < 500
AND roomtype = 'single'
```

HNO	ROOMTYPE	PRICE
30	single	45.00
20	single	70.00
100	single	60.00
110	single	70.00

The query only selects hotels where the weekly price of a single room is less than \$ 500.00. The weekly price is not displayed, because this was not required in the output list.

Find all hotels where the weekly price is less than \$ 500.00 after a price increase of 5 percent:

```
SELECT hno, roomtype, price
FROM room
WHERE (price*1.05)*7 < 500
AND roomtype = 'single'
```

HNO	ROOMTYPE	PRICE
30	single	45.00
100	single	60.00

The price for two days can be calculated in the following way:

```
SELECT hno, roomtype, price*2 two_days
FROM room
WHERE roomtype = 'single'
```

or

```
SELECT hno, roomtype, price+price two_days
FROM room
WHERE roomtype = 'single'
```

HNO	ROOMTYPE	TWO_DAYS
10	single	270.00
30	single	90.00
20	single	140.00
70	single	230.00
80	single	180.00
40	single	170.00
50	single	210.00
60	single	240.00
90	single	180.00
100	single	120.00
110	single	140.00
120	single	160.00
150	single	200.00
130	single	320.00
140	single	250.00

Show the prices for one day, two days and a week:

```
SELECT hno, roomtype, price, price+price two_days, price*7 week
FROM room
WHERE roomtype = 'single'
```

HNO	ROOMTYPE	TWO_DAYS	WEEK
10	single	270.00	945.00
30	single	90.00	315.00
20	single	140.00	490.00
70	single	230.00	805.00
80	single	180.00	630.00
40	single	170.00	595.00
50	single	210.00	735.00
60	single	240.00	840.00
90	single	180.00	630.00
100	single	120.00	420.00
110	single	140.00	490.00
120	single	160.00	560.00
150	single	200.00	700.00
130	single	320.00	1120.00
140	single	250.00	875.00

Show the customers with positive accounts and add the constant 'CREDIT\_BALANCE':

```
SELECT name, account, 'CREDIT_BALANCE' comment
FROM customer
WHERE account > 0
```

NAME	ACCOUNT	COMMENT
Porter	100.00	CREDIT_BALANCE
DATASOFT	4813.50	CREDIT_BALANCE
Peters	650.00	CREDIT_BALANCE
TOOLware	3770.50	CREDIT_BALANCE
Brown	440.00	CREDIT_BALANCE

Let the sum of the positive accounts be \$ 9774.00. Show the percentage portions of all customers (with positive accounts), rounded to the second place to the right of the decimal point, and sorted in descending order:

```
SELECT name, account, fixed (account/9774.00*100,5,2) percentage
FROM customer
WHERE account > 0
ORDER BY 3 DESC
```

NAME	ACCOUNT	PERCENTAGE
DATASOFT	4813.50	49.25
TOOLware	3770.50	38.58
Peters	650.00	6.65
Brown	440.00	4.50
Porter	100.00	1.02

## 4.9 Columns with Set Functions

ADABAS has some set functions that operate in columns on several rows. These functions are called briefly set functions.

The functions	produce the values
SUM	the sum total
MIN	the minimum
AVG	the average
MAX	the maximum
COUNT (distinct...)	the number of different values
COUNT (*)	the number of all values
STDDEV	the standard deviation
VARIANCE	the variance

NULL values are not included in the calculation, except for COUNT (\*).

The query

```
SELECT SUM(account) sum_account, MIN(account) min_account,
       FIXED (AVG(account),7,2) avg_account,
       MAX(account) max_account, COUNT(*) number
FROM customer
WHERE city = 'Los Angeles'
```

gives the table:

SUM ACCOUNT	MIN ACCOUNT	AVG ACCOUNT	MAX ACCOUNT	NUMBER
-164.17	-4167.79	-23.45	3770.50	7

'SUM(account)' produces the sum of the values stored in the account column for all selected rows.



How many customers are there?

```
SELECT COUNT(*) number  
FROM customer
```

NUMBER
15

In how many cities do these customers live?

```
SELECT COUNT(DISTINCT city) number_of_cities  
FROM customer
```

NUMBER_OF_CITIES
6

How many customers who are companies are taken into account? What is the average value of their accounts?

```
SELECT COUNT(*) number, FIXED (AVG(account),7,2) avg_account  
FROM customer  
WHERE firstname IS NULL
```

NUMBER	AVG_ACCOUNT
2	4292.00

Set functions operate on groups of numbers, but they return only one value. The result therefore consists of one row. Whenever a set function is used in a query, a set function must be applied to any other column of the query. This is not true for a column used for grouping by means of GROUP BY. In such a case, the value of the set function is determined for every group.

This is noted as:

```
function name (expression)
```

The parentheses are needed. In most cases, 'expression' is only a column name, but it can also be

- an arithmetic expression,
- any expression formed by other functions,
- a constant,
- DISTINCT with column name.



## 4.10 Operations with Character Strings

There are various functions for the processing of CHAR-type result values.

### Value Code Conversion

For different kinds of applications, it might be necessary to recode values, because different operators demand it.

#### CHR - NUM

Conversion of character values into numbers and vice versa.

Example: Searching for patterns in a sequence of digits

```
SELECT * FROM hotel
WHERE CHR(zip) LIKE '43%'
```

#### UPPER - LOWER

Conversion of lower case letters into upper case letters and vice versa (thus unifying the kind of letters).

Example: Searching for values irrespective of small or capital letters

```
SELECT * FROM hotel
WHERE UPPER (name) = 'STAR'
```

---

ASCII - EBCDIC	<p>Conversion of an ASCII-coded character into the corresponding EBCDIC representation and vice versa.</p> <p>Example:           Outputting a sorted result table in EBCDIC order</p> <pre>SELECT EBCDIC (name) FROM WHERE ... ORDER BY 1</pre> <p>See also the section 'Ordering Rows' (2.5).</p>
MAPCHAR	<p>Conversion of country-specific letters into another representation. This is done to enable a useful alphabetical order.</p>
ALPHA	<p>Conversion of an ASCII- or EBCDIC-coded character into another one- or two-character representation defined in the DEFAULTMAP. This function internally uses MAPCHAR and additionally converts the character into an upper case character. ALPHA also influenceS the sort sequence.</p>
HEX	<p>Conversion into the hexadecimal representation.</p>
CHAR	<p>Conversion of a date or time value into a character string.</p>
SOUNDEX	<p>Conversion of a character string into a representation generated by the so-called SOUNDEX algorithm. This representation can be used if the exact spelling of a term is not known ("sounds like").</p>

**VALUE**

Conversion of a NULL value into another value. In the following example, the title does not occur in the output list, and for companies, the word 'Company' is to be output as 'firstname' instead of the NULL value.

```
SELECT VALUE (firstname, 'Company')
           firstname, name
FROM customer
```

**DECODE**

Conversion of expressions according to their values. The designations for roomtype are to be replaced on output by a code defined by using the DECODE function.

```
SELECT hno, price,
       DECODE (roomtype, 'single', 1,
                'double', 2,
                'suite' , 3)
           code_of_room
FROM room
```

### Concatenating Two CHAR Columns

CHAR columns can be concatenated by using the operator '&'.

```
SELECT name, city&', '&state&' '&chr(zip) address  
FROM customer
```

The numeric column 'zip' must be converted first, before the statement can be executed.

NAME	ADDRESS
Porter	New York, NY 10580
DATASOFT	Dallas, TX 75243
Randolph	Los Angeles, CA 90018
Smith	Los Angeles, CA 90011
Brown	Hollywood, CA 90029
Jackson	Washington, DC 20037
Howe	New York, NY 10019
Miller	Chicago, IL 60601
Peters	Los Angeles, CA 90013
Baker	Los Angeles, CA 90008
Jenkins	Los Angeles, CA 90005
Adams	Los Angeles, CA 90014
Griffith	New York, NY 10575
TOOLware	Los Angeles, CA 90002
Brown	Hollywood, CA 90025

### Eliminating and Inserting Characters

If values of two columns of different lengths are to be concatenated and the blanks between them are to be reduced, the function TRIM can be used.

In the following example, account values greater than zero are multiplied by 1.5 in order to convert them into DM amounts.

```
SELECT name, TRIM (CHR (account * 1.5)) & ' DM' account
FROM account
WHERE account > 0.00
```

NAME	ACCOUNT
Porter	150.000 DM
DATASOFT	7220.250 DM
Peters	975.000 DM
TOOLware	5655.750 DM
Brown	660.000 DM

RTRIM can be used to truncate one or more characters specified as second argument from the right of a column value.

The function LTRIM truncates characters from the left of a value.

### Shortening Values

In the following example, the SUBSTR function is used to reduce the firstname to one letter, to provide it with a period and a blank and then to concatenate it with the name.

```
SELECT SUBSTR(firstname,1,1)&'. ' &name name, city
      FROM customer
      WHERE firstname IS NOT NULL
```

NAME	CITY
J. Porter	New York
?. DATASOFT	Dallas
P. Brown	Hollywood
M. Jackson	Washington
G. Howe	New York
F. Miller	Chicago
M. Griffith	New York
R. Brown	Hollywood

### Displacing Values

The LFILL or RFILL function can be used to pad CHAR-type values with any character up to a specified length.

```
SELECT LFILL(firstname,' ',8) firstname, name, city
FROM customer
WHERE firstname IS NOT NULL AND
city = 'Los Angeles'
```

FIRSTNAME	NAME	CITY
Martin	Randolph	Los Angeles
Sally	Smith	Los Angeles
Joseph	Peters	Los Angeles
Susan	Baker	Los Angeles
Anthony	Jenkins	Los Angeles
Thomas	Adams	Los Angeles

EXPAND expands a character string by a specified number of blanks.

### Creating Bar Charts

```
SELECT name, account, LPAD(' ',TRUNC(account/100),'*',50) graph
FROM customer
WHERE account > 0
ORDER BY account DESC
```

NAME	ACCOUNT	GRAPH
DATASOFT	4813.50	*****
TOOLware	3770.50	*****
Peters	650.00	*****
Brown	440.00	****
Porter	100.00	*

LPAD inserts asterisks just *before* the first parameter (here a blank). This is done according to the number given by account divided by 100.

RPAD inserts asterisks to the *right* of the blank.



### Determining the Number of Characters

The table 'customer' is sorted according to the lengths of the names. If names have the same length, they are sorted in alphabetically ascending order.

```
SELECT name, LENGTH(name) length_of_name
FROM customer
ORDER BY length, name
```

NAME	LENGTH_OF_NAME
Howe	4
Adams	5
Baker	5
Brown	5
Brown	5
Smith	5
Miller	6
Peters	6
Porter	6
Jackson	7
Jenkins	7
DATASOFT	8
Griffith	8
Randolph	8
TOOLware	8

### Replacements in Character Strings

The function REPLACE substitutes one character string for another string in the specified column.

In the following example, the abbreviated notation of street shall be replaced by the complete spelling to obtain a uniform representation.

```
SELECT hno, city, state, zip
       REPLACE (address, 'tr.', 'treet') address
FROM hotel
WHERE address LIKE '%tr'
```

HNO	CITY	STATE	ZIP	ADDRESS
10	Detroit	MI	48226	155 Beechwood Street
20	Cincinnati	OH	45211	1499 Grove Street
80	Chicago	IL	60607	12 Barnard Street
100	Dallas	TX	75225	1980 34th Street
110	New York	NY	10570	111 78th Street

TRANSLATE replaces single letters in the specified column by other letters. For each occurrence, the *i*th letter of the first character string is replaced by the *i*th letter of the second character string. The following statement performs such a replacement that - as we must admit - does not make much sense in the present case.

```
SELECT name, TRANSLATE (name, 'ae', 'oi') name_new
FROM customer
WHERE firstname IS NOT NULL AND
city = 'Los Angeles'
```

NAME	NAME NEW
Randolph	Rondolph
Smith	Smith
Peters	Pitirs
Baker	Bokir
Jenkins	Jinkins
Adams	Adoms

**Notational Checks**

The function INITCAP, issued on a character string, produces a character string where the first letter of each word is output as upper case character and the following letters are output as lower case characters. This function can be used, e.g., to unify the notation of names.

```
SELECT name, INITCAP (name) name_new
FROM customer
WHERE firstname IS NULL
```

NAME	NAME_NEW
DATASOFT TOOLware	Datasoft Toolware

### Searching the Position of Character Strings

A specified substring is to be searched in a character string; the function INDEX produces the position of the substring.

The character string where the search is to take place is specified as the first parameter of INDEX. The second parameter specifies the substring to be searched. In an optional third parameter, the start position for the search can be specified; in an optional fourth parameter, the occurrence of the substring to be searched can be specified.

In the following example, the position of the character string 'ow' is to be determined in all customer names.

```
SELECT name, INDEX(name, 'ow') position_ow
FROM customer
```

NAME	POSITION_OW
Porter	0
DATASOFT	0
Randolph	0
Smith	0
Brown	3
Jackson	0
Howe	2
Miller	0
Peters	0
Baker	0
Jenkins	0
Adams	0
Griffith	0
TOOLware	3
Brown	

### Determining Minima and Maxima in Character Strings

The functions MIN and MAX that have already been presented for numeric values can also be applied to character strings.

The following statement finds that customer in a city whose name begins with the 'smallest' letter relative to the selected code (ASCII or EBCDIC). If the initial letters are identical, the comparison is extended to the characters following thereafter. To obtain a useful alphabetical order, especially of umlauts, the function MAPCHAR should be used.

```
SELECT city, MIN (name) min_name
FROM customer
GROUP BY city
```

CITY	MIN_NAME
Chicago	Miller
Dallas	DATASOFT
Hollywood	Brown
Los Angeles	Adams
New York	Griffith
Washington	Jackson

The functions GREATEST and LEAST can be used to search the greatest or smallest value from a list of specified values. These functions can be applied to any data type. For example, GREATEST ('Baker', 'Baxter', 'Barker') would produce 'Baxter' as the result, because the 'x' as the first differing letter is the 'greatest' letter in alphabetical order.

## 5 Subqueries

### 5.1 IN, ALL, ANY, EXISTS

The query

```
SELECT name, city, state, zip, account
FROM customer
WHERE account =
  (SELECT MAX(account)
   FROM customer)
```

gives the table:

NAME	CITY	STATE	ZIP	ACCOUNT
DATASOFT	Dallas	TX	75243	4813.50

The query finds the customer with the largest account.

The second SELECT statement, enclosed in parentheses, is called a subquery. It is a completely self-contained SQL query. Its execution produces a value or a set of values as part of the main query. The main query is completed by these values in order to make a valid command of it.

First the above query finds a value, i.e. the largest account of any customer. This amount becomes part of the conditional expression 'account = (...)'. The main query selects the customer whose account satisfies this condition.

It is important that the subquery only produces one value. If it generates more values, the conditional expression 'account = (more than one value)' is no longer a

valid expression. The subquery generally selects one column; and in most cases, it only returns one row of this column.



But it happens sometimes that a subquery finds values from more than one row.

The following example contains a condition with the operator IN: It finds all customers and their accounts which are equal to the accounts of the customers in New York. As a list after IN may contain more than one value, the inner, subordinate query can return more than one value (but from one and the same column).

```
SELECT cno, name, city, account
FROM customer
WHERE account IN
  (SELECT DISTINCT account
   FROM customer
   WHERE city = 'New York')
```

The statement

```
SELECT city, FIXED (AVG(account),7,2)
FROM customer
GROUP BY city
HAVING AVG(account) >= ALL
  (SELECT AVG(account)
   FROM customer
   GROUP BY city)
```

has as result:

the city or cities with the largest average account. 'ALL' means that the wanted city has an average account equal to or greater than any other average account found by the subquery.

The statement

```
SELECT name, city
FROM hotel
WHERE name = ANY
      (SELECT city
       FROM hotel)
```

has as result:

a list of hotels (and their cities) which have the same names as any cities in the base table. The subquery produces a list of city names which is then used for the comparison of the hotel names.

The statement

```
SELECT * FROM customer
WHERE EXISTS
      (SELECT * FROM reservation
       WHERE customer.cno = reservation.cno)
```

makes the condition:

Select reservations only if there is one or more reservations.

The customer number establishes a connection between the tables 'customer' and 'reservation'. Thus the example anticipates the description given in section 6.

ALL or ANY are used whenever a subquery produces either more than one value or no value at all and when this is taken into account in a condition which usually requires just one value.

- 'WHERE value = ALL(result of the subquery)' is true if the condition 'WHERE value = result' is true for *every* result produced by the subquery. Operations other than '=' are possible. If one of the results is NULL, the result of the condition with ALL is unknown.

The example above where ALL is used produces the cities the average account of which is the largest average account of all cities.

- 'WHERE value = ANY(result of the subquery)' is true if the condition 'WHERE value = result' is true for *any* result produced by the subquery.

The example above where ANY is used produces names of hotels which are identical with any city names.

EXISTS is used when the subquery does not need to produce a value, but only is to find out whether there is a row that meets a specific condition.

Find the average accounts for all cities where the average account is greater than that of all customers:

```
SELECT city, FIXED (AVG(account),7,2)
FROM customer
GROUP BY city
HAVING AVG(account) >
(SELECT AVG(account)
FROM customer)
```

Show all customers who made a reservation for the 04/12/1995 as the day of arrival:

```
SELECT name
FROM customer
WHERE cno IN
```

```
(SELECT DISTINCT cno  
  FROM reservation  
 WHERE arrival = '04/12/1995')
```

Show all customers who made no reservation at all:

```
SELECT cno, name
FROM customer
WHERE cno = ALL
      (SELECT cno
       FROM reservation
       WHERE arrival > '03/13/1995')
```

Subqueries can be nested: i.e., it is possible to subordinate queries to subqueries.

Find all hotels of the city where a customer has the largest account of all customers:

```
SELECT name, city
FROM hotel
WHERE city =
      (SELECT city
       FROM customer
       WHERE account =
            (SELECT MAX(account)
             FROM customer))
```

## 5.2 Correlation

Subqueries can be used to formulate conditions for the selection of rows which are not to be applied to all table rows, but only to a group of rows.

But first the example of a query which finds the customer who has the largest account of an entire table:

```
SELECT name, city, state, zip, account
FROM customer
WHERE account =
  (SELECT MAX(account)
   FROM customer)
```

result:

NAME	CITY	STATE	ZIP	ACCOUNT
DATASOFT	Dallas	TX	75243	4813.50

A subquery is called a correlated subquery when it refers to columns of outer tables. Non-correlated subqueries are evaluated only once. Correlated subqueries are evaluated for each row of the outer table; in nested subqueries, the evaluation starts with the innermost subquery and ends with the outermost query.

The following is an example of correlation. It shows the customers who have the largest accounts in their respective cities:

```
SELECT name, city, account
FROM customer this_customer
WHERE account =
  (SELECT MAX(account)
   FROM customer
   WHERE city = this_customer.city)
ORDER BY city
```

result:

NAME	CITY	ACCOUNT
Miller	Chicago	0.00
DATASOFT	Dallas	4813.50
Brown	Hollywood	440.00
TOOLware	Los Angeles	3770.50
Porter	New York	100.00
Jackson	Washington	0.00

'this\_customer' in the example is called a 'reference name'.

As the same table is accessed in the inner and outer query of the above example, a reference name must be specified. The task of the reference name is to associate or 'correlate' a row from the result of the main query with a value in the conditional statement.

The following is a detailed description of the example:

<code>SELECT name, city, account</code>	Find name, city, and account
<code>FROM customer this_customer</code>	of the table 'customer' and call it 'this_customer'.
<code>WHERE account =</code>	Keep the row where the account is equal to the following subquery:
<code>(SELECT MAX(account)</code>	(Start of the subquery)
<code>FROM customer</code>	Find the maximum account of the table 'customer'
<code>WHERE city =</code>	where the city name is equal to the city name of the row selected above.
<code>this_customer.city)</code>	
<code>ORDER BY city</code>	Order the rows alphabetically by the city names.



Another example of the usage of correlation variables performs a grouping according to 'roomtype'. Those hotels are searched where the prices are less than the average price of the respective type of room.

```
SELECT hno, roomtype, price
  FROM room x
 WHERE price <
    (SELECT AVG (price)
      FROM room
     WHERE roomtype = x.roomtype)
```

HNO	ROOMTYPE	PRICE
30	double	80.00
20	double	100.00
80	double	150.00
40	double	140.00
90	double	150.00
100	double	100.00
110	double	130.00
120	double	140.00
30	single	45.00
20	single	70.00
80	single	90.00
40	single	85.00
90	single	90.00
100	single	60.00
110	single	70.00
120	single	80.00
80	suite	400.00
90	suite	300.00
120	suite	350.00
150	suite	450.00



## 6Columns from Two and More Tables

### 6.1Information from Two Tables

The following statement asks whether there is a reservation for the customer 'Porter' and, if so, for what date.

To answer this question, two tables must be searched. The customer name is stored in the table 'customer', the reservation date in the table 'reservation'. The connection between these tables is established by the customer number which occurs in both tables. Such a statement is called a join.

```
SELECT reservation.rno, customer.name, reservation.arrival,  
       departure  
FROM customer, reservation  
WHERE customer.name = 'Porter' AND  
       customer.cno = reservation.cno
```

RNO	NAME	ARRIVAL	DEPARTURE
100	Porter	11/13/1994	11/15/1994
110	Porter	12/24/1994	01/06/1995

Syntax note:

Joining the two customer numbers in the WHERE clause establishes the necessary bridge. Possible operators are '=' (equal to), '<' (less than), '<=' (less than or equal to), '>' (greater than), '>=' (greater than or equal to) and '<>' (not equal to).

The whole query can be read in the following way:

Find all pairs of rows from 'customer' and 'reservation', where the customer number is the same in both halves of rows and the name in the customer table is 'Porter'. Select the reservation number, the customer name, and the traveling dates from the concatenated row.



The next example joins the two tables without any restriction to a particular person. Therefore the previously found person is contained in this result.

```
SELECT reservation.rno, customer.cno, name,  
       reservation.arrival, departure  
FROM customer, reservation  
WHERE customer.cno = reservation.cno
```

RNO	CNO	NAME	ARRIVAL	DEPARTURE
100	3000	Porter	11/13/1994	11/15/1994
110	3000	Porter	12/24/1994	01/06/1995
180	3100	DATASOFT	12/23/1994	01/08/1995
120	3200	Randolph	11/14/1994	11/18/1994
150	3600	Howe	03/14/1995	03/24/1995
130	3900	Baker	02/01/1995	02/03/1995
160	4100	Adams	04/12/1994	04/15/1994
140	4300	TOOLware	04/12/1994	04/30/1994
190	4300	TOOLware	11/14/1994	11/17/1994
170	4400	Brown	09/01/1994	09/03/1994

If columns in two different tables have the same name, the table name must be specified in front of the column name. The two names are connected by a dot. To improve the legibility of statements, it is recommended to place the table name also in front of unique column names and to connect the names by a dot.

## 6.2 Information from Three Tables

The first example finds all hotels and the respective cities where the customer 'Porter' has booked a room. Three tables must be joined for this purpose.

```
SELECT customer.name, reservation.rno,  
       name_of_hotel = hotel.name, city  
FROM customer, reservation, hotel  
WHERE customer.name = 'Porter' AND  
       customer.cno = reservation.cno AND  
       reservation.hno = hotel.hno
```

NAME	RNO	NAME_OF_HOTEL	CITY
Porter	100	Midtown	Chicago
Porter	110	Dallas	Dallas

Show all customers and the cities where they have booked a hotel. Values from the table 'reservation' are not displayed in this case. But the table is needed to retrieve the hotel number.

```
SELECT customer.name, hotel.city
      FROM customer, reservation, hotel
     WHERE customer.cno = reservation.cno AND
           reservation.hno = hotel.hno
```

NAME	CITY
Porter	Chicago
Porter	Dallas
DATASOFT	Los Angeles
Randolph	Chicago
Howe	New York
Baker	New York
Adams	New York
TOOLware	Chicago
TOOLware	Washington
Brown	Santa Clara

### 6.3 Outer Join

Find all hotels in Chicago for which reservations exist. These hotels are displayed in the result table with their respective numbers and names.

```
SELECT hotel.hno, hotel.name, reservation.rno
FROM hotel, reservation
WHERE hotel.city = 'Chicago' AND
      hotel.hno = reservation.hno
```

HNO	NAME	RNO
50	Lake Michigan	120
80	Midtown	100
80	Midtown	140

To have a list of all hotels in Chicago displayed, irrespective of the availability of one or more reservations, a so-called 'outer join' is used.

A result table is generated which - like the first example - shows all hotels in Chicago with the corresponding reservations, also containing the hotels for which no reservations have been made. The missing entries for the reservation numbers are set to the NULL value.



The outer join is denoted by the (+) operator.

```
SELECT hotel.hno, hotel.name, reservation.rno
FROM hotel, reservation
WHERE hotel.city = 'Chicago' AND
hotel.hno = reservation.hno (+)
```

HNO	NAME	RNO
40	Eight Avenue	?
50	Lake Michigan	120
80	Midtown	100
80	Midtown	140

## 7Set Operations

This section presents some possibilities of relating results from several result tables to each other and generating a new result table from these related results. For this purpose, operations which, in a similar form, are known from the set theory are used.

### 7.1UNION

The UNION statement enables the user to generate a union from result tables produced by two or more SELECTs.

In the simplest case, two result tables which have been generated from the same base table can be related to each other.

For example, if all customers living either in Los Angeles or in New York are to be found and if UNION is used, this can be represented in the following way:

```
SELECT title, firstname, name, city
      FROM customer
      WHERE city = 'Los Angeles'
UNION
SELECT title, firstname, name, city
      FROM customer
      WHERE city = 'New York'
```

This result could also have been obtained by a simple SELECT and an OR specification:

```
SELECT title, firstname, name, city
FROM customer
WHERE city = 'Los Angeles' OR city = 'New York'
```

Beyond this, the UNION statement allows result tables to be combined that have been generated from different tables.

It must be ensured then that the data types of the respective output columns can be compared to each other. Equality is not required, because the maximum length is used, if this should be necessary. Consequently, CHAR (10) and CHAR (15) columns can be combined with each other, because the length is automatically extended to CHAR (15).

The effect of UNION, INTERSECT, and EXCEPT is shown with and without ALL, using the column 'city' of the tables 'hotel' and 'customer'.

To get a better overview of the retrieved results, the example shall only refer to cities located in the states greater than or equal to IL.

The examples are based on the following result tables:

```
SELECT cities_of_customers = city
FROM customer
WHERE zip < 50000
```

```
SELECT cities_of_hotels = city
FROM hotel
WHERE zip < 50000
```

CITIES OF CUSTOMER
New York
Washington
New York
New York

CITIES OF HOTELS
Detroit
Cincinnati
New York
New York
Washington

Find both the cities where the customers live and the cities where the hotels are located. For this purpose, a union is formed across the tables 'customer' and 'hotel'.

```
SELECT city
  FROM customer
 WHERE zip < 50000
 UNION
SELECT city
  FROM hotel
 WHERE zip < 50000
```

CITY
Detroit
New York
Washington
Cincinnati

It is obvious here that repeatedly occurring cities are output only once. The database issues an implicit DISTINCT for UNION.

To obtain all cities with all their occurrences, the statement UNION ALL can be specified.

```
SELECT city
  FROM customer
 WHERE zip < 50000
 UNION ALL
SELECT city
  FROM hotel
 WHERE zip < 50000
```

CITY
Detroit
New York
New York
New York
New York
New York
Washington
Washington
Cincinnati

## 7.2 INTERSECT

A subset relation can be established using the statement INTERSECT.

All cities are to be found that occur in the table 'customer' as well as in the table 'hotel'. Without the additional specification of ALL, an implicit DISTINCT is issued again.

```
SELECT city
      FROM customer
      WHERE zip < 50000
INTERSECT
SELECT city
      FROM hotel
      WHERE zip < 50000
```

CITY
New York
Washington

Values repeatedly occurring in the subset are displayed by using the following statement. The result values only occur as often as the values from the two tables 'customer' and 'hotel' have a counterpart in the corresponding other table.

```
SELECT city
      FROM customer
      WHERE zip < 50000
      INTERSECT ALL
SELECT city
      FROM hotel
      WHERE zip < 50000
```

CITY
New York
New York
Washington



## 7.3 EXCEPT

The EXCEPT clause allows results of one result table to be subtracted from another result table.

Show all cities that are found in the table 'hotel', but are not contained in the result table of 'customer'.

```
SELECT city
  FROM hotel
 WHERE zip < 50000
EXCEPT
SELECT city
  FROM customer
 WHERE zip < 50000
```

CITY
Detroit
Cincinnati

The sequence of SELECT statements is not arbitrary, unlike for UNION and INTERSECT.

If all result rows are to be retrieved that are contained in 'customer' but not in 'hotel', the message 'Row not found' is output. The cities Dallas and New York have a counter part in the table 'hotel'. Therefore they are not returned. In this case, it is not important that the customer table contains one more entry for New York than the table 'hotel'.

Before EXCEPT comes into effect, an implicit DISTINCT is issued on the tables.

If each occurrence of the rows found in the particular result table are to be taken into account, EXCEPT ALL must be specified.

```
SELECT city
  FROM customer
 WHERE zip < 50000
EXCEPT ALL
SELECT city
  FROM hotel
 WHERE zip < 50000
```

CITY
New York

'ALL' has prevented DISTINCT from being issued in this case as well. As 'NY' occurs once more in the table 'customer' than in the table 'hotel', that value is kept as the result.

To get a clear notion of this statement, one could think of the coinciding values of both tables be 'checked off', the remaining values of the first table forming the result.

In our example:

Two entries of 'Dallas' 'neutralize each other'.



## 8 Data Definition and Entries

### 8.1 Creating a Table

The following statement defines a table 'person':

```
CREATE TABLE person
  (cno          FIXED(4),    - The cno is a four-digit number.
   firstname    CHAR(7),    - The first name is seven and
   name         CHAR(8),    the name is eight characters
                           long.
   account      FIXED(7,2)) - The account contains up to five
                           digits for dollar and two digits
                           for cent amounts.
```

The statement has the following structure:

It consists of the keywords CREATE TABLE, followed by the table name, and (in parentheses) a list of column identifiers separated by commas. The optional PRIMARY KEY clause can be used to define whether a primary key is assigned to the table (a detailed description of the primary key concept is given in section 9); or referential integrity constraints are defined (cf. section 8.7).

Each column is identified by

- the column name,
- the data types FIXED, FLOAT, CHAR (BYTE), VARCHAR, LONG, DATE, TIME, TIMESTAMP or BOOLEAN or some less usual types that are mapped to the types specified so far, or a domain name (cf. section 8.13),
- an optional column restriction defined as CONSTRAINT and NOT NULL,
- an optional default value which is automatically entered into the column when the user does not specify an explicit value for the column.

### *Temporary Tables*

A temporary table is a special kind of table. Temporary tables only exist during the session of a user and are deleted afterwards. They are denoted by TEMP that precedes their name.

## **8.2 Data Types in a Table**

FIXED represents fixed point numbers. The first figure within the parentheses specifies the maximum number of digits; the second figure indicates the number of places to the right of the decimal point. If there is no second figure, zero is assumed. The maximum number of all digits is 18.

The data type FLOAT identifies positive or negative floating point numbers within a range of values of (10 to the power -64) to (10 to the power +62). The numeric string can have up to 18 digits in length.

CHAR is the data type for character strings of a maximum length of 4000. These strings can be stored in ASCII or EBCDIC code. If no specification is made, the strings are stored in the code of the particular computer. CHAR fields with lengths greater than 30 are internally stored as variable length fields.

To obtain a code-neutral storage, the field can be defined with the type CHAR BYTE.

VARCHAR defines CHAR fields with the additional property of being internally stored in variable length in any case.

LONG specifies an alphanumeric column of any length. LONG columns can be processed by INSERT, UPDATE, and DELETE statements. However, there are some restrictions for the processing. LONG columns can be used, for example, for the storage of long texts.

DATE is the data type for date values. The representation depends on the selected date format. In SQL statements, every user must use the date format he selected for the representation. The current date can be retrieved by the function DATE.

If TIME was specified, time values can be stored in such a column. Also in this case, the representation depends on the chosen format. The current time can be retrieved by the function TIME.

TIMESTAMP allows a timestamp value to be stored that consists of a date and a time including microseconds. The current value can be retrieved using the function TIMESTAMP.

BOOLEAN defines a column that can only receive the values TRUE and FALSE.

There are some additional data types that are internally mapped to the data types specified above. These are the following types: INTEGER, SMALLINT, DECIMAL, FLOAT, DOUBLE PRECISION, REAL, and LONG VARCHAR. Their meaning is not described in detail.

### 8.3 Column Constraints

The range of values of a column's data type can be restricted further by using so-called constraints. There is a distinction between simple and complex constraints.

*Simple* constraints are conditions that only refer to the one column to be defined.

An upper and lower bound for the values to be entered can be defined, for example, using BETWEEN <lower bound> AND <upper bound>. The IN predicate allows the valid values to be listed.

```
cno      FIXED (4)   CONSTRAINT cno BETWEEN 1 AND 9999
title    CHAR  (5)   CONSTRAINT title IN ('Mr', 'Mrs', 'Comp')
account  FIXED (7,2) CONSTRAINT account > -10000 AND
                                account < 10000
```

Specifying NOT NULL has the effect that a column *must* be provided with a value. Such a column is called a mandatory column. Without a NOT NULL specification, the column is optional. A constraint defined for a column implicitly means that the NULL value cannot be entered.

*Complex* constraints are those referring to more columns.

In a table 'reservation', for example, where the days of arrival and departure are stored, it could be useful to check whether the arrival is before the departure.

```
arrival   DATE NOT NULL
departure DATE CONSTRAINT departure > arrival
```

This condition can be extended at will:

```
departure DATE CONSTRAINT departure > arrival
                                AND departure < '12/31/1995'
```





Except for very few conditions, everything that could also be a valid search condition can be formulated in a CONSTRAINT definition. In principle, any number of columns can be addressed. But it should be taken into account that the additional checks performed in the case that modifications have been made to this table decrease the system performance. Several conditions can be connected with the operators AND, OR, and NOT.

## 8.4 Replicated Tables

ADABAS D allows the installation of a distributed database.

The data can be distributed to different SERVERNODEs. A local database is called a SERVERDB. The whole database can consist of one or more SERVERDBs.

In a distributed configuration, data can be created in different ways:

### *Not replicated (local)*

If a CREATE TABLE statement is specified (as described in section 8.1), the description of the table structure as well as the contents of the table are stored on a SERVERNODE. If this data is accessed from another SERVERNODE, a communication must be established within the network.

### *Replicated*

WITH REPLICATION added to the CREATE TABLE statement has the effect that the description of the structure and the inserted data are available on all SERVERNODEs in the form of replications. Read operations are accelerated in this way. Modifying operations must be performed on all SERVERNODEs and are therefore more complicated and consequently, more 'expensive'.

## 8.5 Inserting Rows

The following statement inserts a row into the table 'person':

```
INSERT person VALUES (3391, 'Fred', 'Marx', -4.75)
```

Syntactic variants are:

```
INSERT INTO person VALUES (3395, 'Charles', 'Brand', -4913.00)
```

```
INSERT INTO person (cno, name, firstname, account)
VALUES (3396, 'Higgins', 'Mark', -2.19)
```

```
INSERT INTO person SET account = -640.30,
                        name    = 'Brown',
                        firstname = 'Hank',
                        cno      = 3393
```

If no column names are specified, the sequence of values must correspond to the sequence of the columns in the definition. Both sequences must be identical in length and data type. Undefined values can be written as NULL.

```
SELECT cno, firstname, name, account
FROM person
```

CNO	FIRSTNAME	NAME	ACCOUNT
3391	Fred	Marx	-4.75
3395	Charles	Brand	-4913.00
3396	Mark	Higgins	-2.19
3393	Hank	Brown	-640.30

The table 'person' can be easily included into the table 'customer' after having been extended by the mandatory columns 'city', 'state', and 'zip':

```
INSERT INTO customer
  SELECT cno, 'Mr', firstname, name, 'New York', 'NY', '10573'
         account
  FROM person
```

## 8.6 Updating Rows

This command updates values in the table 'person'. Small negative accounts are set to 0.

```
UPDATE person SET account = 0
      WHERE account > -10 AND
            account <= 0

SELECT cno, firstname, name, account
      FROM person
```

CNO	FIRSTNAME	NAME	ACCOUNT
3391	Fred	Marx	0.00
3395	Charles	Brand	-4913.00
3396	Mark	Higgins	0.00
3393	Hank	Brown	-640.30

Note that *all* rows of the table are updated when a WHERE condition is missing.

The general format of the UPDATE statement is:

```
UPDATE.....  table name
SET.....      modification of the column values
WHERE.....     which rows
```

## 8.7Deleting Rows

This statement deletes all people having an account equal to 0. The system determines and eliminates the relevant rows.

```
DELETE FROM person
      WHERE account = 0
```

The result is:

```
SELECT cno, firstname, name, account
      FROM person
```

CNO	FIRSTNAME	NAME	ACCOUNT
3395	Charles	Brand	-4913.00
3393	Hank	Brown	-640.30

Note here, too, that *all* rows of the table are deleted when a WHERE condition is missing.

The general format of the DELETE statement is:

```
DELETE FROM.....  table name
WHERE.....         which rows
```

## 8.8 Relations between Tables

An interrelation between two tables can be defined which will influence the modification of rows.

table customer

cno	title	.....
-----	-------	-------

table reservation

rno	cno	hno	...
-----	-----	-----	-----

Such a relation is called a referential integrity constraint and determined when the table 'reservation' is defined. An interrelation between the table 'reservation' and 'customer' is defined by assigning a so-called foreign key to 'reservation'. This foreign key corresponds to the key of 'customer'.

```
CREATE TABLE reservation
(rno      FIXED (4) KEY,
 cno      FIXED (4),
 hno      FIXED (4),
 roomtype CHAR (6),
 arrival  DATE,
 departure DATE,
 FOREIGN  KEY (cno)
 REFERENCES customer
 ON DELETE CASCADE)
```

The name of the referential integrity constraint can be specified by the user after the keywords FOREIGN KEY, or it will be given by the database itself. In the latter case, the names of the tables concerned are concatenated - separated by an underscore - (up to a maximum length of 18 characters).

The relation defined in the example above gets the name 'customer\_reservation'.

A keyword to be specified after DELETE can be used to determine what is to be done with depending values when rows are being deleted.

When rows of the table 'customer' are being deleted, it must be ensured that there are no reservations any more for this table. If there are some, the user can choose among different possibilities:

He specifies

```
ON DELETE RESTRICT
```

in the statement. Then he gets a warning and can act accordingly.

He requires that the corresponding reservation rows are implicitly deleted as well.

```
ON DELETE CASCADE
```

But he can also achieve that the customer number, which has become meaningless, is set to the NULL value or a default value in the affected rows of the reservation table:

```
ON DELETE SET NULL  
ON DELETE SET DEFAULT
```

The relation defined above is also supervised the other way round. Inserting a new reservation or updating an existing reservation for which no customer exists is prevented irrespective of the DELETE rule.

A special ALTER TABLE statement can be used to delete a referential integrity constraint.

The relation between the tables 'reservation' and 'customer' is no longer desired and must therefore be removed again:

```
ALTER TABLE reservation DROP FOREIGN KEY customer_reservation
```



## 8.9 Updating Column Definitions

The table definition can be altered while the database is operative.

The following statement inserts two further columns into the table 'person'. First these columns contain the NULL value in each row.

```
ALTER TABLE person
  ADD (phone FIXED(8),
       city  CHAR(10))
```

The new columns are immediately available for the processing:

```
UPDATE person SET phone = 670900, city = 'Los Angeles'
  WHERE name = 'Brand'
```

```
SELECT cno, firstname, name, city, account, phone
  FROM person
```

CNO	FIRSTNAME	NAME	CITY	ACCOUNT	PHONE
3395	Charles	Brand	Hollywood	-4913.00	670900
3393	Hank	Brown	?	-640.30	?

In a similar way, columns or integrity rules can be removed. The column 'account' is to be dropped from the table:

```
ALTER TABLE person DROP (account)
```

Beyond this, column definitions can be altered. The new definition will only be executed if the values that are already stored correspond to it. The column 'name', for example, can be extended from a length of 8 to a length of 10 characters:

```
ALTER TABLE person COLUMN name CHAR (10)
```

## 8.10 Short Names for a Table

This statement generates the additional name `NEGATIVE` for the table `'person'`. This name can be written anywhere instead of the old name. Since it is a user-specific name, it has the advantage that it can be specified without user identification.

```
CREATE SYNONYM negative FOR person
```

Several synonyms can be defined for one table. They can neither be used nor be seen by other users.

Synonyms are particularly useful for abbreviating long or complicated table names. Especially, when accessing tables of other users, the user name specification before the table name can be omitted.

## 8.11 Creating Views

A view has the effect of a window laid over an existing base table allowing parts of it to be seen, others to be hidden.

The following statement defines a view that includes all columns (cno, title, name, firstname, city, state, zip, and account), but only those rows that have a value greater than or equal to 0 in the column 'account'.

```
CREATE VIEW v1 AS
  SELECT *
    FROM customer
   WHERE account >= 0
```

The statement

```
SELECT * FROM v1
```

produces:

CNO	TITLE	NAME	FIRSTNAME	CITY	STATE	ZIP	ACCOUNT
3000	Mrs	Porter	Jenny	New York	NY	10580	100.00
3100	Comp	DATASOFT	?	Dallas	TX	75243	4813.50
3200	Mr	Randolph	Martin	Los Angeles	CA	90018	0.00
3300	Mrs	Smith	Sally	Los Angeles	CA	90011	0.00
3400	Mr	Brown	Peter	Hollywood	CA	90029	0.00
3500	Mr	Jackson	Michael	Washington	DC	20037	0.00
3700	Mr	Miller	Frank	Chicago	IL	60601	0.00
3800	Mr	Peters	Joseph	Los Angeles	CA	90013	650.00
4000	Mr	Jenkins	Anthony	Los Angeles	CA	90005	0.00
4200	Mr	Griffith	Mark	New York	NY	10575	0.00
4300	Comp	TOOLware	?	Los Angeles	CA	90002	3770.50
4400	Mrs	Brown	Rose	Hollywood	CA	90025	440.00

Columns can be renamed and rearranged by using a view. Several tables can be joined. Each SELECT statement that does not contain ORDER BY can be used to define a view. A view name can be used in SELECT, INSERT, UPDATE, and DELETE statements. For an INSERT, columns missing in the view will be completed either by default values defined for this purpose or by the NULL value. But this cannot be done for every view; there are different restrictions.

A view has two purposes:

Long SELECT statements can be reduced.

Unimportant or confidential data can be hidden.

## 8.12 Creating Snapshot Tables

A view table, as described in the previous section, provides a logical view to data that is physically stored in a base table. A snapshot table is described like a view, but it allows *copies* of partial data to be created from a base table.

A so-called snapshot is created from a described section of a table. This snapshot is physically stored in the database.

Modifications to the table on which the snapshot is based are not automatically done to the snapshot. The modifications can be done asynchronously either by copying the whole data or by using the modification log, the so-called snapshot log. The execution of the modifications are initiated by the REFRESH statement.

In snapshot tables, only SELECTS are possible. INSERT, UPDATE or DELETE statements are not allowed.

A section from the table 'customer' is to be recorded in a snapshot table. A snapshot log is generated for the table:

```
CREATE SNAPSHOT snap1 AS
  SELECT cno,name,firstname,city
  FROM customer
```

```
CREATE SNAPSHOT LOG ON customer
```

The table 'customer' on which the snapshot is based is then modified by removing all customers living in Los Angeles from the table.

```
DELETE FROM customer
  WHERE city = 'Los Angeles'
```

A SELECT issued on the snapshot table then produces the following result:

CNO	NAME	FIRSTNAME	CITY
3 000	Porter	Jenny	New York
3 100	DATASOF	?	Dallas
3 200	Randolph	Martin	Los Angeles
3 300	Smith	Sally	Los Angeles
3 400	Brown	Peter	Hollywood
3 500	Jackson	Michael	Washington
3 600	Howe	George	New York
3 700	Miller	Frank	Chicago
3 800	Peters	Joseph	Los Angeles
3 900	Baker	Susan	Los Angeles
4 000	Jenkins	Anthony	Los Angeles
4 100	Adams	Thomas	Los Angeles
4 200	Griffith	Mark	New York
4 300	TOOLware	?	Los Angeles
4 400	Brown	Rose	Hollywood

The command

```
REFRESH SNAPSHOT snap1
```

has the effect that the modifications on the table 'customer' are also executed on the table 'snap1'.

```
SELECT * from snap1
```

CNO	NAME	FIRSTNAME	CITY
3 000	Porter	Jenny	New York
3 100	DATASOFT	?	Dallas
3 400	Brown	Peter	Hollywood
3 500	Jackson	Michael	Washington
3 600	Howe	George	New York
3 700	Miller	Frank	Chicago
4 200	Griffith	Mark	New York
4 400	Brown	Rose	Hollywood

## 8.13 Creating an Index

Since all table columns are treated in the same way, each can be used as a search criterion. This does not mean, however, that they are equally efficient.

If a column is preferred for making conditions for a search or an update, it is recommendable to create an index file for it. This file helps to find the table rows more quickly.

A single-column index on the column 'name' of the table 'customer' is to be created. Two syntactical variants are provided for this purpose. In the first case, an index named 'name\_idx' is created; in the second case, the unnamed index is identified using the table name and column name.

```
CREATE INDEX name_idx on customer (name)

CREATE INDEX customer.name
```

An index can refer to more columns; then it is called a multiple index and must be named.

```
CREATE INDEX name_idx on customer (name, firstname)
```

To create an index which, like the key (see section 9), ensures uniqueness, the keyword UNIQUE must be specified.

The definition of a UNIQUE index can be included in the table definition. Although uniqueness would be too restrictive for the name, the two examples above would look like this:

```
CREATE TABLE customer (cno FIXED (4) ...
                        title
                        name CHAR (8) UNIQUE,
                        firstname
                        )

CREATE TABLE customer (cno FIXED (4) ...
                        title
                        name
                        firstname ...

                        UNIQUE (name,firstname)
                        )
```

In the second case, the database generates a name of its own, INDEX01.



## 8.14 Creating Domains

Before defining a table, single columns can be defined separately as domains.

It would have been possible to say for the table 'person':

```
CREATE DOMAIN name CHAR(8).
```

This domain can be used later for the table definition:

```
CREATE TABLE person
(cno          FIXED(4),      - The cno is a four-digit number.
firstname     name,          - The first name and the name
name          name,          - are eight characters long.
account       FIXED(7,2))    - The account contains up to five
                              digits for dollar and two digits
                              for cent amounts.
```

A DOMAIN definition can include definitions for default values and constraints. Within a CONSTRAINT definition, the domain name must be used instead of the column name (cf. section 8.3).

When default values are used, it must be ensured that these meet the constraints.

Another column is to be inserted into the table 'person' in order to record the birthday. This definition is also done by using a predefined DOMAIN. The current date is taken as the default value. A constraint is defined that the person was not born before the 01/01/1880.

```
CREATE DOMAIN birthday_dom
DATE DEFAULT DATE
CONSTRAINT birthday_dom > '01/01/1880' AND
birthday_dom <= DATE
```

## 8.15DB Procedures

DB procedures are ADABASIC programs which are called from an application program like *one* SQL statement. A DB procedure can contain several SQL statements and the application developer can use the control structures provided by ADABASIC. So, for example, loops or branches can be programmed within a DB procedure.

Input parameters and output parameters can be defined which can be used to pass own values to the DB procedure or to return results from the DB procedure.

DB procedures are directly executed in the address space of the kernel, not that of the user. Thus communication overhead between application and database kernel is saved. The performance can be improved especially for client server configurations, because the kernel operates on another computer than the application.

With DB procedures, a common SQL access layer is provided on the server side which can be used by all application programs. Thus the user is given simple access to complex database operations.

Possible fields of applications are, e.g., the formulation of complex integrity rules for validity checks of values and the provision of operations on application objects (abstract data types). Modifications to these rules or operations can be done at a central place and must no longer be done for each individual application. This allows programs to be more clearly structured and to be more easily maintained.

Access can be simplified to a still greater extent by granting privileges. It is sufficient to grant the call privilege for a DB procedure. No privileges for the database objects addressed by the DB procedure are needed in addition.

## 8.16 Triggers

In contrast to DB procedures which must be called *explicitly* from an application programming language, triggers are *implicitly* started after executing an INSERT, UPDATE, or DELETE statement on a base table.

If a trigger is defined for a base table and this trigger is to be started for each INSERT, for example, then the actions determined within this trigger are automatically executed whenever a new entry is made to that table. Preliminary conditions can be defined for the execution of a trigger. UPDATE triggers can be formulated only for the update of individual columns.

A useful case of application for triggers is to check, in addition to the domain definition of a value to be inserted, whether this value is appropriate for the entry. Any complex checks can be started in background, even related to other tables.

Triggers can also incite derived database modifications for one or another row; they can even contain complicated rules for access controls.

Before performing an UPDATE or DELETE, the old value can be saved into another table for statistical purposes. They can be processed there at a later point in time.

These examples illustrate that both the new as well as the old value can be accessed when programming a trigger. This is denoted by the keywords NEW and OLD in the trigger program.

Triggers are defined in two steps by using the programming language ADABASIC.

First, the actions to be started when calling a trigger are defined in an ADABASIC program. Such a program is denoted by the keyword TRIGGER. The control structures provided by ADABASIC can be used for this purpose; the same constraints apply that are valid for DB procedures (cf. the ADABASIC manual).

Second, the trigger must be associated with a base table and with individual columns, if desired, as well as with an action, such as INSERT, UPDATE, or

DELETE. Column values are passed to the trigger by using defined formal parameters.

An input menu is offered by the ADABASIC workbench for the definition of formal parameters. This menu prompts for the required specifications. Conditions can be specified there that restrict the execution of a trigger. For example, it can be determined that the trigger is only to be started when the input value is greater than a given value.

To be able to define a trigger for a table, one must be the owner of the table and must have the right to execute actions defined within a trigger. The same rules for access control that are valid for DB procedures are valid for the triggers.

If the execution of a trigger fails, the corresponding INSERT, UPDATE, or DELETE statement is also cancelled by a rollback.

A trigger can call other triggers implicitly and DB procedures explicitly. Like a DB procedure, a trigger is directly performed in the address space of the database kernel. This has the advantage that communication overhead is saved, especially in client server configurations.

## 8.17DB Functions

ADABASIC allows the effects of SQL statements to be extended by user-specific DB functions. Thus more applications logic can be shifted to the database server.

DB functions are defined in the programming language ADABASIC. They can be used like predefined functions in the SELECT list of a command, in the WHERE qualification or within the SET clause of the UPDATE statement.

DB functions are specialized procedures that have any number of input parameters but only one output parameter. The output parameter represents the result of the function, thus defining the data type of the function's result.

SQL statements are not allowed within DB functions. It must be ensured that no name of a predefined function is used when naming a DB function.

## 8.18 Dropping Objects

The following statements drop the following objects from the database:

DROP TABLE	person	removes the table 'person'.
DROP VIEW	v1	removes the view 'v1'.
DROP SNAPSHOT	snap1	removes the snapshot table 'snap1'.
DROP SNAPSHOT LOG ON	customer	removes the snapshot log.
DROP SYNONYM	negative 'negative'.	removes the additional name
DROP INDEX	customer.name	removes the index 'customer.name'.
DROP DOMAIN	name	removes the domain 'name'.
DROP TRIGGER	account_statistics	removes a trigger.
DROP TABLE	removes a table definition together with its contents and depending objects such as views, synonyms or indexes. So BE CAREFUL! The statement can only be executed by the owner of a table.	
DROP VIEW	removes the user window on table contents. The contents themselves remain untouched.	
DROP SNAPSHOT	removes the copy (snapshot) of the contents of a table.	
DROP SNAPSHOT LOG ON	removes the modification log (snapshot log) of a table.	

DROP SYNONYM	removes the additional name of a table. The original definition and the contents of the table are kept.
DROP INDEX	removes the index file which was created in order to increase the system performance. This is the only effect.
DROP DOMAIN	removes the definition of a domain. Table definitions which used these domains are kept. Of course, CREATE TABLE statements containing dropped domain definitions can no longer be issued.
DROP TRIGGER	removes a trigger. The ADABASIC procedure where these processing steps are defined is kept.



## 8.19 Transactions

ADABAS is a transaction-oriented database system with reset facility.

This means that a safety mechanism runs parallel to ADABAS. This mechanism allows any modifications made to the database to be cancelled. The statement

```
ROLLBACK WORK
```

restores the status as it was at the beginning of a session. While working with the database, the user has the possibility to define the point in time up to which modifications to the database are reset. This point in time can be redefined by using the statement

```
COMMIT WORK
```

Whenever COMMIT WORK is issued, a 'transaction' is concluded.

This means in practice that the activities can be tried out first and then, when the user is satisfied with the results, the status reached so far can be saved permanently using COMMIT WORK, thus becoming the starting point of the next transaction. If, on the other hand, errors and undesired effects have occurred, these can be cancelled using ROLLBACK WORK, thus returning to the last point where the database has been saved.

Example:

```
COMMIT WORK
```

```
SELECT cno, firstname, name, account  
FROM person
```

CNO	FIRSTNAME	NAME	ACCOUNT
3391	Fred	Marx	-4.75
3395	Charles	Brand	-4913.00
3396	Mark	Higgins	-2.19
3393	Hank	Brown	-640.30

```
UPDATE person SET account = 0
      WHERE account > -10 AND
      account <= 0
```

```
SELECT cno, firstname, name, account
      FROM person
```

CNO	FIRSTNAME	NAME	ACCOUNT
3391	Fred	Marx	0.00
3395	Charles	Brand	-4913.00
3396	Mark	Higgins	0.00
3393	Hank	Brown	-640.30

ROLLBACK WORK

```
SELECT cno, firstname, name, account
      FROM person
```

CNO	FIRSTNAME	NAME	ACCOUNT
3391	Fred	Marx	-4.75
3395	Charles	Brand	-4913.00
3396	Mark	Higgins	-2.19
3393	Hank	Brown	-640.30

So-called subtransactions can be used within transactions to define logical units. Subtransactions can be nested.

A subtransaction is opened with SUBTRANS BEGIN; if it was successful, it is closed with SUBTRANS END. If the modifications are not desired, they can be rolled back by using SUBTRANS ROLLBACK.

Related to the transaction concept is the locking concept.

There are two kinds of locks. Setting a read lock allows all users to read the locked object, but prevents them from modifying it. To alter an object, the user needs a write lock which prevents other users from read- or write-accessing the object.

Locks are generally implicitly set by the system and are kept up to the transaction's end. But the user can also set them explicitly for certain areas by using the LOCK statement. These locks are held up to the transaction's end, as well.

The user indicates the lock mode he wants to work with, specifying an isolation level. ADABAS distinguishes the isolation levels 0, 10, 15, 20, 30. The larger the number, the better the consistency ensured for the read data. But this will have a negative effect on the possible concurrency of data accesses. See also the ADABAS Reference manual.

In precompiler programs, the CONNECT statement can be used to make known the LOCK mode.

## 9 Accessing Single Rows

### 9.1 Key Concept

For some applications, it is convenient to clearly identify the rows of a table by one or more columns to be able to process the table, for example, in a fixed sequential order. To do so, a primary key is defined. The column names that are to form the key of the table are preceded by 'PRIMARY KEY'. The input values of the key columns thus defined must not be NULL.

The rows of the table 'city' are to be uniquely identified by the zip code. A single-column primary key is created for this purpose. Within the table definition, the zip code need not necessarily be defined as the first column.

```
CREATE TABLE city    (name          CHAR(15),
                      state         CHAR(2) NOT NULL,
                      zip           FIXED(5),
                      PRIMARY KEY   (zip))
```

There is a syntactic variant for defining a table with a key. The key columns are provided with the attribute KEY. In the CREATE TABLE statement, they must be listed as the first columns in the order of definition.

```
CREATE TABLE city    (zip          FIXED(5) KEY,
                      name         CHAR(15),
                      state        CHAR(2) NOT NULL)
```

Rows are inserted in the same way as into a base table without a KEY definition. The following examples refer to the above table definition:

```
INSERT city
VALUES (90013, 'Los Angeles', 'CA')
INSERT city
VALUES (75243, 'Dallas', 'TX')
INSERT city
VALUES (90018, 'Los Angeles', 'CA')
INSERT city
VALUES (10580, 'New York', 'NY')
INSERT city
VALUES (20037, 'Washington', 'DC')
INSERT city
VALUES (90029, 'Hollywood', 'CA')
```

Another attempt to insert a city having the zip code 90013 results in an error message. The KEY definition ensures the uniqueness of the column.

## 9.2 Direct Access Using a Key

A table with key columns can, of course, be processed with the usual SELECT statement.

```
SELECT zip, name, state
FROM city
ORDER BY state
```

ZIP	NAME	STATE
90013	Los Angeles	CA
90018	Los Angeles	CA
90020	Hollywood	CA
20037	Washington	DC
10580	New York	NY
75243	Dallas	TX

Key columns can also be used for direct access to a single row. The key is specified in the WHERE condition by the additional keyword KEY.

```
SELECT DIRECT name FROM city KEY zip = 10580
```

NAME
New York

The single row access is very efficient. It does not need an index.

A single row access, however, is not only performed for a SELECT DIRECT specification.

If SELECT...INTO is used in a program and if it is ensured at the same time that the WHERE clause contains equality conditions for all key columns, the efficient single row access is performed in this case, too.

For more detailed information, refer to the Reference manual or to the Precompiler manuals.

### 9.3 Position-based Access Using a Key

One can think of the rows as being stored in sequential key order. If one key of this sequence is known, it has the function of a pointer. Starting from this pointer, the next or the previous row can be found. The pointer need not necessarily denote an existing key.

```
SELECT NEXT name FROM city KEY zip = 10580
```

NAME
Washington

```
SELECT PREV name FROM city KEY zip = 90013
```

NAME
Dallas

Note: If an index was created for any non-key column, such as 'name' (see section 'Creating an Index'), the stacked job processing of a table can even be performed using this 'secondary key column':

```
SELECT NEXT zip, name FROM city  
INDEX name = 'New York'  
KEY zip = 10580
```

PLZ	NAME
10580	New York



The very first or the very last row can be selected with:

```
SELECT FIRST name FROM city
```

NAME
New York

```
SELECT LAST name FROM city
```

NAME
Hollywood

The effect of `SELECT FIRST` corresponds to the search for the row which in ascending order directly follows the smallest key consisting of binary zeros. `SELECT LAST`, on the other hand, retrieves the row with the largest key value.

## 9.4 Conditional Position-based Access

Position-based access in key sequence need not scan all stored rows. The sequence of rows can be limited in two ways:

- The usage of a view name has the effect that not all rows of a particular table are visible.
- A WHERE condition defines additional conditions on the row in question.

```
SELECT LAST name FROM city  
      WHERE state = 'CA'
```

NAME
Hollywood

```
SELECT NEXT name FROM city KEY zip = 20037  
      WHERE state = 'CA'
```

NAME
Los Angeles

## 9.5 Multiple Key

A key need not consist of only one column, as it was used in the above example. It can consist of up to 127 columns which must not exceed a length of 255 characters, altogether.

Usually, keys are not constructed of more than five columns, because otherwise the user who must enter unique values would lose the overview.

Now let us see the slightly modified table 'customer'. First, the department where a particular customer is recorded is to be recorded. Within such a department, the customers are numbered by hundreds; this will be checked using the modulo function.

```
CREATE TABLE customer (deptno  FIXED (4) KEY
                        CONSTRAINT deptno BETWEEN 1 and 999,
                        cno      FIXED (4) KEY
                        CONSTRAINT cno BETWEEN 1 AND 9999
                                AND cno MOD 100 = 0,
                        title    ...
                        name     ...
                        )

INSERT customer
VALUES (100, 3000, 'Mrs', ...)

INSERT customer
VALUES (100, 3100, 'Mr', ...)

INSERT customer
VALUES (100, 3200, 'Mr', ...)

INSERT customer
VALUES (200, 3000, 'Comp', ...)

INSERT customer
VALUES (200, 3000, 'Mrs', ...)
```

The entered values illustrate that neither the department number nor the customer number could guarantee uniqueness. But as a multiple key, these two numbers ensure that the customers are uniquely identified.

The order of the columns specified after the PRIMARY KEY clause defines the key sequence.

The direct access to a customer must be formulated in the following way:

```
SELECT DIRECT title, name
FROM customer
WHERE deptno = 100, cno = 3100
```



## 10 Authorization

### 10.1 User Classes

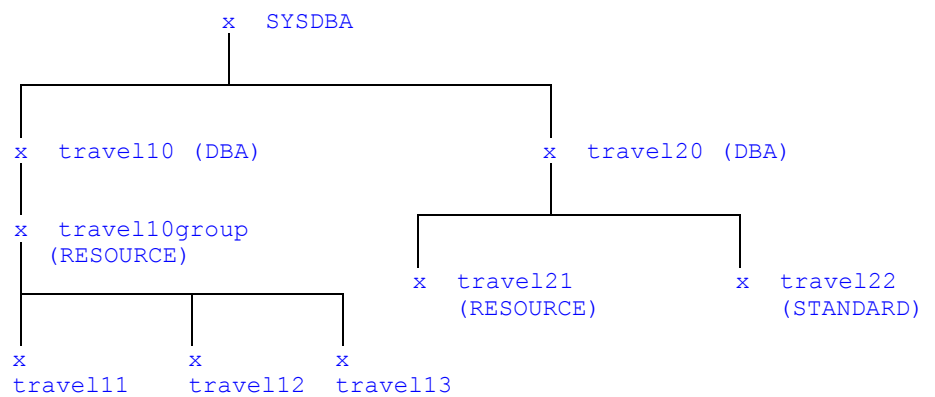
ADABAS distinguishes six user classes:

CONTROLUSER	<ul style="list-style-type: none"><li>- is created when the system is generated.</li><li>- performs all administrative tasks in WARM and COLD mode by using the ADABAS component CONTROL.</li></ul>
SYSDBA (system database administrator)	<ul style="list-style-type: none"><li>- is created when the system is generated.</li><li>- is unique within the net when a distributed database system is concerned.</li><li>- can install the user classes DBA, RESOURCE, and STANDARD.</li><li>- can create any kind of data and grant owner privileges for it.</li></ul>
DOMAINUSER	<ul style="list-style-type: none"><li>- maintains the data dictionary.</li><li>- The name is always DOMAIN.</li></ul>
DBAs (database administrators)	<ul style="list-style-type: none"><li>- are installed by the SYSDBA.</li><li>- can create any kind of data and grant user privileges for it.</li><li>- can install the user classes RESOURCE and STANDARD.</li></ul>
RESOURCE users	<ul style="list-style-type: none"><li>- are installed by the SYSDBA or DBA.</li><li>- can operate on any kind of data.</li><li>- can create local and temporary data.</li></ul>
STANDARD users	<ul style="list-style-type: none"><li>- are installed by the SYSDBA or DBA.</li><li>- can operate on any kind of data.</li><li>- can create temporary data.</li></ul>

RESOURCE and STANDARD users can be combined to form the respective usergroups.

## 10.2 The SYSDBA Installs DBAs

Let the following organizational structure be planned for the application 'travel agency'. Only one SERVERDB shall be available.



First the SYSDBA creates two DBAs. The tables used in the examples shall be assigned to these DBAs at a later point in time.

The user 'travel10' shall be able to connect simultaneously and repeatedly to the database from different terminals. This is obtained by the specification 'NOT EXCLUSIVE'. The user 'travel20' shall not be allowed to do this.

```
CREATE USER travel10
  PASSWORD t10
  DBA NOT EXCLUSIVE
```

```
CREATE USER travel20
  PASSWORD t20
  DBA
```

If a distributed database is concerned, one could also think of another constellation. There are two SERVERNODEs, with one SERVERDB each, which form the distributed database.

The respective SYSDBAs create the users 'travel10' and 'travel20' on different SERVERNODEs.

SERVERDB1:

```
x  SYSDBA1
|
x  travel10 (DBA)
|
x  travel10group
   (RESOURCE)
|
...
```

SERVERDB2:

```
x  SYSDBA2
|
x  travel20 (DBA)
|
...
```



### 10.3 The DBAs Install Additional Users

The database administrators create the subordinate users and the tables associated with their fields of application. They also have the task to assign privileges for their data.

To install the user structure, the DBA 'travel10' creates the usergroup 'travel10group'. The users belonging to this group are created afterwards.

```
CREATE USERGROUP travel10group
RESOURCE
```

```
CREATE USER travel11
PASSWORD t11
USERGROUP travel10group
```

```
CREATE USER travel12
PASSWORD t12
USERGROUP travel10group
```

```
CREATE USER travel13
PASSWORD t13
USERGROUP travel10group
```

The groupname cannot be used for connecting to the database. This must be done using the name of the group member. Objects, such as tables, however, are stored with the groupname.

'travel20' creates single users who do not belong to a group. 'travel21' and 'travel22' cannot be combined to form a group, because they are differently authorized. 'travel22' as STANDARD user shall not be authorized to create own tables.

```
CREATE USER travel21
PASSWORD t21
RESOURCE
```

```
CREATE USER travel22
PASSWORD t22
STANDARD
```

The management of the table 'customer' is assigned to the DBA 'travel10'. The DBA decides that the members of his usergroup shall be authorized to maintain the customer data, but not to alter the table structure.

```
GRANT SELECT, UPDATE, DELETE, INSERT
      ON customer
      TO travel10group
```

All users stored in the system shall be authorized to read the data. The granting of the privileges need not be done for each single user, it can be done using the keyword PUBLIC.

```
GRANT SELECT
      ON customer
      TO PUBLIC
```

'travel20' is responsible for the tables 'hotel', 'room', and 'reservation'. This DBA grants different privileges for his tables to his users 'travel21' and 'travel22'.

```
GRANT SELECT, UPDATE, DELETE, INSERT
      ON hotel, room, reservation
      TO travel21
```

```
GRANT SELECT, UPDATE
      ON hotel
      TO travel22
```

The usergroup shall obtain the right to operate on these three tables. 'travel20' wants to entrust the granting of the privileges to the DBA to whom the group belongs. Therefore 'travel20' enables 'travel10' to operate on his tables as well as to grant privileges for them. The specification WITH GRANT OPTION informs the system about that.

```
GRANT ALL
      ON hotel, reservation, customer
      TO travel10
      WITH GRANT OPTION
```

'travel10' received all rights for the table 'hotel' and passes some of them on to the usergroup. As this table does not belong to him, the name specification must be completed by the owner name.

```
GRANT SELECT, UPDATE, DELETE, INSERT  
ON travel20.hotel  
TO travel10group
```

```
GRANT SELECT  
ON travel20.reservation  
TO travel10group
```

## 10.4 Modifying and Removing User Profiles

Granted privileges can be revoked. The owner of the data and the users who passed privileges granted to them to other users are authorized to do that.

The following statement issued by the user 'travel20' revokes the update right on the table 'hotel' from the user 'travel22':

```
REVOKE UPDATE ON hotel FROM travel22
```

'travel20', for example, does no longer want that 'travel10' has all privileges and the authorization to grant them to other users. The REVOKE statement has the effect that all privileges that 'travel10' granted on the table 'hotel' to other users are automatically revoked from the users. In our example, the entire authorization for 'hotel' is revoked from the usergroup:

```
REVOKE ALL ON hotel FROM travel10
```

User and usergroup can be removed from the database. The DBA who installed the user or usergroup is authorized to do so.

```
DROP USER travel22  
DROP USERGROUP travel10group
```

DROP USER and DROP USERGROUP remove the specified user entry from the database catalog. They implicitly remove all the rights associated with this entry (password, privileges), any existing private data and rights granted for it and - for usergroups - all pertinent users.

Organizational modifications in the application 'travel agency' can be carried out with the relevant authorization statements. The user 'travel21' becomes DBA in order to be able to install users of his own.

1) Altering the user mode:

```
ALTER USER travel21 DBA
```

This statement must be performed by the SYSDBA, because he is the only one who can create DBAs. It is also possible to alter the status of a usergroup.

2) Altering the password:

- by the SYSDBA using

```
ALTER PASSWORD travel10 TO f4ffg
```

- by the respective user using

```
ALTER PASSWORD t10 f4ffg
```

3) Revoking granted privileges

```
REVOKE SELECT          ON customer FROM PUBLIC
```

```
REVOKE INSERT, DELETE ON customer FROM travel10group
```

```
REVOKE ALL             ON hotel FROM travel10
```

4) Granting privileges:

```
GRANT INDEX, REFERENCES, ALTER ON hotel TO travel21
```

## 11 Catalog Information

All objects stored in the database as well as the relationships between these objects are stored in the so-called database catalog. ADABAS provides different ways to retrieve information from the catalog.

On the one hand, there is DOMAIN which can be used to create and maintain objects in addition to making queries to the catalog. On the other hand, information about the objects is stored in various system tables which can be accessed in the usual way by using SELECT statements. Owner of these tables is the special user 'domain'.

Since the number of system tables and relationships represented in these tables is very large, we show a few important queries in the following sections. The exact structure of the system tables is contained in the Reference manual.

## 11.1 Tables

The first object to be dealt with is the object 'table'. The user 'travel10' wants to retrieve information about the definition of his table 'customer'. For this purpose, the user 'travel10' formulates a SELECT on the system table 'domain.columns':

```
SELECT *  
      FROM DOMAIN.COLUMNS  
WHERE tablename = 'CUSTOMER'  
AND   owner = 'TRAVEL10'  
ORDER BY POS
```

This statement shows all column names, their data types, the column lengths, the decimal representation for numeric fields, and the privileges of the requesting user. The displayed list contains indications of whether mandatory columns are concerned and constraints or default values are defined.

OWNER	TABlename	COLUMNNAME	MOD	DATATYPE	CODETYPE
TRAVEL10	CUSTOMER	CNO	KEY	FIXED	
TRAVEL10	CUSTOMER	TITLE	OPT	CHAR	ASCII
TRAVEL10	CUSTOMER	NAME	MAN	CHAR	ASCII
TRAVEL10	CUSTOMER	FIRSTNAME	OPT	CHAR	ASCII
TRAVEL10	CUSTOMER	CITY	MAN	CHAR	ASCII
TRAVEL10	CUSTOMER	STATE	MAN	CHAR	ASCII
TRAVEL10	CUSTOMER	ZIP	OPT	FIXED	
TRAVEL10	CUSTOMER	ACCOUNT	OPT	FIXED	

	LEN	DEC	COLUMNPRIVILEGES	DEFAULT	
	4	0	SEL+UPD+	?	
	5		SEL+UPD+	?	
	8		SEL+UPD+	?	
	7		SEL+UPD+	?	
	11		SEL+UPD+	?	
	2		SEL+UPD+	?	
	5	0	SEL+UPD+	?	
	7	2	SEL+UPD+	?	

The query on the table 'domain.columns' produced information about a specific table. The next query is to produce a list of tables for which the user 'travel10' has access privileges. The listed indications are essentially the owner of the table, the privileges, and statistical time information.

```
SELECT *
FROM DOMAIN.TABLES
ORDER BY owner,tablename
```

This query can be restricted according to desired criteria, e.g., to the own tables:



```
SELECT *  
  FROM DOMAIN.TABLES  
 WHERE owner = 'REISEN10'  
 ORDER BY owner,tablename
```

## 11.2 Domains

To display a specific domain definition, use the statement

```
SELECT definition
FROM domain.domains
WHERE domainname = 'NAME'
```

DEFINITION
CREATE DOMAIN NAME CHAR (8)

Detailed information about a domain is displayed if not only the column 'definition' is selected but also any information contained in the table 'domain.domains' is retrieved:

```
SELECT *
FROM domain.domains
WHERE domainname = 'NAME'
```

If no restriction for a certain domain name is specified, a list is displayed containing all domain definitions available for the definition of tables:

```
SELECT *
FROM domain.domains
```

### 11.3 Constraints

If conditions restricting the range of values were defined, the user can display the corresponding definitions using

```
SELECT definition
FROM domain.constraints
WHERE tablename = 'CUSTOMER'
```

The user specifies the table name and column name.

DEFINITION
TITLE IN ('Mr','Mrs','Comp')

To see a list of all constraint definitions, specify:

```
SELECT *
FROM domain.constraints
```

## 11.4 Views

Now the definitions of views are to be displayed. From the table 'domain.defs', a user can retrieve view definitions that relate to tables for which he has privileges.

Let the following view be defined:

```
CREATE VIEW v1 AS SELECT *
                    FROM customer WHERE account >= 0

SELECT definition
   FROM domain.viewdefs
  WHERE viewname = 'V1'
```

To see a list of all views, specify:

```
SELECT * FROM domain.views
```

Information similar to that displayed for the select issued on the table 'domain.tables' is shown.

## 11.5 Synonyms

Information about synonyms can be found in the table 'syn\_refs\_tab'. A synonym is *defined* by using it for *referencing* a table. The relationships are shown in the column names by the abbreviations 'def' and 'ref'.

```
CREATE SYNONYM c1 FOR customer
```

```
SELECT * FROM syn_refs_tab
```

DEFOBJTYPE	DEFOWNER	DEFSYNONYMNAME	RELTTYPE
SYNONYM	TRAVEL10	C1	REFERS

	REFOBJTYPE	REFOwner	REFTABLENAME	
	TABLE	TRAVEL10	CUSTOMER	

...

Restrict the SELECT list to produce a clearer output list of the synonyms by giving the columns easier names. At the same time, the columns are renamed.

```
SELECT defsynonymname synonymname,  
       refowner owner,  
       reftablename tablename  
FROM domain.syn_refs_tab  
WHERE defsynonymname = 'NEGATIVE'
```

SYNONYMNAME	OWNER	TABLERNAME
NEGATIVE	TRAVEL10	PERSON

## 11.6 Primary Keys

Information about the structure of the primary key can be retrieved from the table 'domain.columns'. In the column 'keypos', all key columns of tables contain an entry not equal to the NULL value.

Since a key can consist of several columns, the key columns should be ordered when being retrieved.

```
SELECT columnname,mode,datatype,codetype,len,dec,
       columnprivileges,default,keypos
FROM domain.columns
WHERE keypos IS NOT NULL
ORDER BY keypos
```

COLUMNNAME	MODE	DATATYPE	CODETYPE	LEN	DEC
CNO	KEY	FIXED		4	0

	COLUMNPRIVILEGES	DEFAULT	KEYPOS
	SEL+UPD+	?	1

## 11.7 Indexes

Queries about created indexes are to be started on the table 'domain.ind\_uses\_col'. As described for the synonym definitions, a distinction is made between created objects and objects referencing the created objects.

To obtain a clear display, several columns should be renamed.

For convenient restrictions, a table name or a special index name could be used. Otherwise, a list is displayed containing all indexes created on objects for which the current user has privileges.

```
SELECT defowner owner,
       deftablename tablename,
       defindexname indexname,
       type,
       refcolumnname columnname,
       pos,sort,
       createdate "DATE",
       createtime "TIME"
FROM domain.ind_uses_col
WHERE deftablename = 'CUSTOMER'
ORDER BY owner,tablename,indexname,pos
```

OWNER	TABLERNAME	INDEXNAME	TYPE
TRAVEL10	CUSTOMER	NAME	

COLUMNNAME	POS	SORT	DATE	TIME
NAME		ASC	31.07.1995	12.11.38



## 11.8 Referential Integrity Constraints

The interrelations existing between tables can be retrieved using the following commands. Here it is obvious that you must know the system tables well in order to be able to completely survey the relations between the tables.

A query could look like this:

```
SELECT defowner owner,
       deftablename tablename,
       defcolumnname columnname,
       defkeyname refname,
       refowner,
       reftablename,
       refcolumnname,
       rule,
       createdate "DATE",
       createtime "TIME"
FROM domain.fkc_refs_col
WHERE deftablename = 'CUSTOMER'
```

OWNER	TABLENAME	COLUMNNAME
TRAVEL20	RESERVATION	CNO

	REFNAME	REFOwner	REFTABLENAME
	CUSTOMER_RESERVATION	TRAVEL10	CUSTOMER

	REFCOLUMNNAME	RULE	DATE	TIME
	CNO	DELETE CASCADE	29.06.1995	09.14.04

## 11.9 Privileges

The current user wants information about the privileges he has on his own tables and that of other users.

The display shows the owner, the table and column names, the privileges for these tables, and the users who granted privileges, if any, to the current user. If the user has the right to grant these privileges, a '+' following the corresponding abbreviation indicates this.

The user 'travel10' wants to display such a list of privileges. He enters the following statement (before 'travel20' revokes the privileges from him):

```
SELECT refowner owner,
       reftablename tablename,
       refcolumnname columnname,
       privileges,
       defusername grantor,
FROM domain.usr_uses_col
WHERE refowner like 'TRAVEL*'
```

OWNER	TABLERNAME	COLUMNNAME
TRAVEL10 TRAVEL20	CUSTOMER HOTEL	- ALL COLUMNS - - ALL COLUMNS -

	PRIVILEGES	GRANTOR
	SEL+UPD+DEL+INS+REF+IND+ALT+ SEL+UPD+DEL+INS+REF+IND+ALT+	TRAVEL10 TRAVEL20

The user 'travell10' can see all privileges he has granted directly or indirectly. These privileges are related to the corresponding tables on display.

Indirect granting of privileges means that the current user gave other users the right to grant privileges for his tables to third users. The user who granted a privilege is indicated in the result table as GRANTOR.

'travell10' wants to know the privileges he has granted for the table 'customer'. He enters:

```
SELECT refowner owner,
       reftablename tablename,
       refcolumnname columnname,
       privileges,
       defusername grantor
FROM domain.usr_uses_col
WHERE defusername = 'TRAVEL10'
AND   refowner = 'TRAVEL10'
AND   reftablename = 'CUSTOMER'
```

OWNER	TABLERNAME	COLUMNNAME
TRAVEL10 TRAVEL10	CUSTOMER CUSTOMER	- ALL COLUMNS - - ALL COLUMNS -

	USERNAME	PRIVILEGES	GRANTOR
	PUBLIC TRAVEL10GROUP	SEL SEL UPD DEL INS	TRAVEL10 TRAVEL10

### 11.10 User Profiles

The table 'domain.users' contains comprehensive information about the users available in the system. For example, it can be queried who created which users and whether there are members of a usergroup. Information retrieved can concern the user mode, as well as restrictions for memory and system time. The display also contains the SERVERDB where the user was created and the SERVERNODE where this SERVERDB is located.

```
SELECT * FROM domain.users
```

## 11.11 Administrative Information

ADABAS provides more system tables that are mainly needed for administrative tasks.

Display of a list of the users currently working with the database:

```
SELECT * FROM domain.connectedusers
```

Query to retrieve information about all SERVERDBs of the distributed database. For example, their state and the corresponding SERVERNODE are indicated.

```
SELECT * FROM domain.serverdbs
```

Information about the current state of the runtime environment and the operational database kernel:

```
SELECT * FROM domain.versions
```

Display of the system database administrator of the SERVERDB to which the current user is connected.

```
SELECT sysdba FROM dual
```

## **11.12 Statistical Information**

The database system provides especially the database administrator with statistical information about different fields of the system.

The information is contained in different system tables which belong to the system database administrator (e.g., sysdba.serverdbstatistics, sysdba.configuration, etc.). A detailed description of the possible queries would go beyond the scope of this book.

## 12The Tables Used

The tables used in this manual for the application 'travel agency' are completely listed in this section. The tables concerned are the tables 'customer', 'hotel', 'room', and 'reservation'.

```
CREATE TABLE customer
(cno          FIXED(4) KEY CONSTRAINT cno BETWEEN 1 AND 9999,
 title        CHAR(5)
             CONSTRAINT title IN ('Mr','Mrs','Comp'),
 name         CHAR(8) NOT NULL,
 firstname    CHAR(7),
 city         CHAR(11) NOT NULL,
             CONSTRAINT zip BETWEEN 10000 AND 99999,
 state        CHAR(2),
 zip          FIXED(5),
 account      FIXED(7,2)
             CONSTRAINT account BETWEEN -10000 AND 10000)
```

CNO	TITLE	NAME	FIRSTNAME	CITY	STATE	ZIP	ACCOUNT
3000	Mrs	Porter	Jenny	New York	NY	10580	100.00
3100	Comp	DATASOFT	?	Dallas	TX	75243	4813.50
3200	Mr	Randolph	Martin	Los Angeles	CA	90018	0.00
3300	Mrs	Smith	Sally	Los Angeles	CA	90011	0.00
3400	Mr	Brown	Peter	Hollywood	CA	90029	0.00
3500	Mr	Jackson	Michael	Washington	DC	20037	0.00
3600	Mr	Howe	George	New York	NY	10019	-315.40
3700	Mr	Miller	Frank	Chicago	IL	60601	0.00
3800	Mr	Peters	Joseph	Los Angeles	CA	90013	650.00
3900	Mrs	Baker	Susan	Los Angeles	CA	90008	-4167.79
4000	Mr	Jenkins	Anthony	Los Angeles	CA	90005	0.00
4100	Mr	Adams	Thomas	Los Angeles	CA	90014	-416.88
4200	Mr	Griffith	Mark	New York	NY	10575	0.00
4300	Comp	TOOLware	?	Los Angeles	CA	90002	3770.50
4400	Mrs	Brown	Rose	Hollywood	CA	90025	440.00



```

CREATE TABLE hotel
(hno          FIXED (4) KEY,
              CONSTRAINT hno BETWEEN 1 AND 9999,
 name         CHAR (13) NOT NULL,
 city         CHAR (11) NOT NULL,
 state        CHAR (2)  NOT NULL,
 zip          FIXED (5)
              CONSTRAINT zip BETWEEN 10000 AND 99999,
 address      CHAR (25) NOT NULL)

```

HNO	NAME	CITY	STATE	ZIP	ADDRESS
10	Congress	Detroit	MI	48226	155 Beechwood Str.
20	Long Island	Cincinnati	OH	45211	1499 Grove Street
30	Regency	Portland	OR	97213	477 17th Avenue
40	Eight Avenue	Chicago	IL	60601	112 8th Avenue
50	Lake Michigan	Chicago	IL	60615	354 OAK Terrace
60	Airport	New Orleans	LA	70112	650 C Parkway
70	Empire State	New York	NY	10019	65 Yellowstone Dr.
80	Midtown	Chicago	IL	60607	12 Barnard Street
90	Long Beach	Long Beach	CA	90804	200 Yellowstone Dr.
100	Dallas	Dallas	TX	75225	1980 34th Str.
110	Atlantic	New York	NY	10570	111 78th Str.
120	Sunshine	Los Angeles	CA	90018	35 Broadway
130	Star	Hollywood	CA	90030	13 Beechwood Place
140	River Boat	Washington	DC	20019	788 MAIN STREET
150	Indian Horse	Santa Clara	CA	95054	16 MAIN STREET

```
CREATE TABLE room
(hno      FIXED (4) KEY,
 roomtype CHAR (6)
          CONSTRAINT roomtype IN
          ('single', 'double', 'suite'),
 max_free FIXED (3) CONSTRAINT max_free >= 0,
 price    FIXED (6,2)
          CONSTRAINT price BETWEEN 0.00 AND 1000.00)
```

HNO	ROOMTYPE	MAX_FREE	PRICE
10	single	20	135.00
10	double	45	200.00
30	single	12	45.00
30	double	15	80.00
20	single	10	70.00
20	double	13	100.00
70	single	4	115.00
70	double	11	180.00
80	single	15	90.00
80	double	19	150.00
80	suite	5	400.00
40	single	20	85.00
40	double	35	140.00
50	single	50	105.00
50	double	230	180.00
50	suite	12	500.00
60	single	10	120.00
60	double	39	200.00
60	suite	20	500.00
90	single	45	90.00
90	double	145	150.00
90	suite	60	300.00
100	single	11	60.00
100	double	24	100.00
110	single	2	70.00
110	double	10	130.00
120	single	34	80.00
120	double	78	140.00
120	suite	55	350.00
150	single	44	100.00
150	double	115	190.00
150	suite	6	450.00
130	single	89	160.00
130	double	300	270.00
130	suite	100	700.00
140	single	10	125.00
140	double	9	200.00
140	suite	78	600.00

```

CREATE TABLE reservation
(rno          FIXED(4) KEY
          CONSTRAINT rno BETWEEN 1 AND 9999,
 cno          FIXED(4)
          CONSTRAINT cno BETWEEN 1 AND 9999,
 hno          FIXED(4)
          CONSTRAINT hno BETWEEN 1 AND 9999,
 roomtype     CHAR (6)
          CONSTRAINT roomtype IN
            ('single', 'double', 'suite'),
 arrival      DATE NOT NULL,
 departure    DATE
          CONSTRAINT departure > arrival)

```

RNO	CNO	HNO	ROOMTYPE	ARRIVAL	DEPARTURE
100	3000	80	single	11/13/1994	11/15/1994
110	3000	100	double	12/24/1994	01/06/1995
120	3200	50	suite	11/14/1994	11/18/1994
130	3900	110	single	02/01/1995	02/03/1995
140	4300	80	double	04/12/1994	04/30/1994
150	3600	70	double	03/14/1995	03/24/1995
160	4100	70	single	04/12/1994	04/15/1994
170	4400	150	suite	09/01/1994	09/03/1994
180	3100	120	double	12/23/1994	01/08/1995
190	4300	140	double	11/14/1994	11/17/1994



## 13Index

### - A -

ABS.....	4-7
ACOS.....	4-9
ADABASIC.....	8-22
ADDDATE, example.....	4-10
ADDTIME.....	4-11
administrative information.....	11-17
ALL.....	5-1; 5-2; 5-4
ALPHA.....	4-22
ALTER PASSWORD.....	10-8
ALTER TABLE.....	8-13
ALTER USER.....	10-8
AND.....	3-1; 3-6; 3-7; 3-8; 3-9; 3-18
ANY.....	5-1; 5-4
arithmetic.....	4-4; 4-5; 4-6; 4-7; 4-12; 4-13; 4-15
arithmetic operations.....	4-12
ASC.....	2-8; 2-11
ASCII.....	2-9; 4-22
ASIN.....	4-9
ATAN.....	4-9
AVG.....	3-19; 3-20; 3-21; 3-22; 3-23; 4-17; 4-20

### - B -

bar chart.....	4-28
BETWEEN.....	3-1; 3-8; 3-9; 3-18
bibliography.....	1-2
BOOLEAN.....	8-1; 8-3

### - C -

calculations in result tables.....	4-4
------------------------------------	-----

CASCADE.....	8-12
catalog.....	11-1
catalog information.....	11-2
CEIL.....	4-7
CHAR.....	4-22; 4-27; 8-1; 8-2
CHAR BYTE.....	8-1; 8-2
characters, eliminate, insert.....	4-25
characters, replace.....	4-30
CHR.....	4-21
column.....	2-8
column constraints.....	8-4
columns, drop.....	8-13
columns, insert.....	8-13
columns, order.....	2-4; 2-11
columns, select.....	2-3; 2-4
columns, update.....	8-13
COMMIT WORK.....	8-28
comparison operators.....	3-3
concatenation.....	4-24
conditions.....	3-1
conditions, arithmetic.....	4-13
conditions, constraint.....	8-4
conditions, for groups.....	3-22
conditions, multiple.....	3-6
conditions, negate.....	3-14; 3-15
conditions, parenthesize.....	3-7; 3-14
constant.....	4-16
CONSTRAINT.....	8-1; 8-4; 11-6
constraint definition, display.....	11-6
constraints, for domains.....	8-21
CONTROLUSER.....	10-1
copies of data.....	8-6
copy.....	8-17
correlation.....	5-6
COS.....	4-9
COSH.....	4-9
COT.....	4-9
COUNT.....	4-17

CREATE DOMAIN.....	8-21; 11-5
CREATE INDEX.....	8-19
CREATE SNAPSHOT.....	8-17
CREATE SYNONYM.....	8-14
CREATE TABLE.....	8-1; 8-6
CREATE USER.....	10-3
CREATE USERGROUP.....	10-4
CREATE VIEW.....	8-15; 8-16; 11-7

## **- D -**

data types.....	8-2
database administrator.....	10-1; 10-4
DATE.....	4-10; 4-11; 8-1; 8-3
date functions.....	4-10
DATEDIFF, example.....	4-10
DAY.....	4-11
DAYNAME.....	4-11
DAYOFMONTH.....	4-11
DAYOFWEEK.....	4-11
DAYOFYEAR.....	4-11
DB functions.....	8-25
DB procedures.....	8-22; 8-23
DBA.....	10-1; 10-4
DECIMAL.....	8-3
DECODE.....	4-23
default value.....	8-1
DEFAULTMAP.....	2-9
defaults, for domains.....	8-21
DEGREES.....	4-9
delete.....	8-10
demo samples.....	1-1
DESC.....	2-8; 2-11
DISTINCT.....	2-13; 4-17
distributed database.....	8-6
DIV.....	4-6
domain.....	8-21



domain definition, display.....	11-5
domain, drop.....	8-27
DOMAIN, tool.....	11-1
DOMAINUSER.....	10-1
DOUBLE PRECISION.....	8-3
drop columns.....	8-13
DROP DOMAIN.....	8-26; 8-27
DROP INDEX.....	8-26; 8-27
DROP SNAPSHOT.....	8-26
DROP SNAPSHOT LOG ON.....	8-26
DROP SYNONYM.....	8-26; 8-27
DROP TABLE.....	8-26
DROP TRIGGER.....	8-26; 8-27
DROP USER.....	10-7
DROP USERGROUP.....	10-7
DROP VIEW.....	8-26

## - E -

EBCDIC.....	2-9; 4-22
eliminate.....	2-13; 3-22
eliminate, insert characters.....	4-25
encoding.....	2-9; 4-22
equal.....	3-1; 3-3
ESCAPE.....	3-13
EXCEPT.....	7-8
EXCEPT ALL.....	7-9
EXISTS.....	5-1; 5-3; 5-4
EXP.....	4-8
EXPAND.....	4-27

## - F -

FALSE.....	3-1
FIXED.....	4-7; 4-12; 8-1; 8-2
FLOAT.....	4-12; 8-1; 8-2; 8-3

FLOOR.....	4-7
FOR REUSE.....	2-15
foreign key.....	8-11

## **- G -**

GRANT.....	10-5
GRANTOR.....	11-15
greater.....	3-1; 3-3
GREATEST.....	4-34
GROUP BY.....	3-19; 3-20; 3-21; 3-22
grouping.....	3-19; 3-20; 3-21; 3-22; 4-17; 4-19
groups, conditions.....	3-22

## **- H -**

HAVING.....	3-22
HEX.....	4-22
HOURL.....	4-11

## **- I -**

IN.....	3-1; 3-10; 3-17; 5-1; 5-2
INDEX.....	4-33; 9-5; 11-11
index definition, display.....	11-11
index, create.....	8-19
index, drop.....	8-27
index, unique.....	8-19
INITCAP.....	4-32
insert.....	8-7; 8-13
INTEGER.....	8-3
INTERSECT.....	7-6
INTERSECT ALL.....	7-7
IS NULL.....	2-6; 3-15; 4-20
isolation level.....	8-30

**- J -**

join.....6-1; 6-2; 6-3

**- K -**

keep.....2-15  
key.....9-1; 9-3; 9-5; 11-10; 11-11  
KEY attribute.....9-1  
key, multiple.....9-8

**- L -**

LEAST.....4-34  
LENGTH.....4-29  
less.....3-1; 3-3  
LFILL.....4-27  
LIKE.....3-1; 3-11; 3-12; 3-13; 3-16  
LN.....4-8  
local data.....8-6  
LOCK.....8-30  
lock mode.....8-30  
locking.....8-30  
LOG.....4-8  
logarithm.....4-8  
LONG.....8-1; 8-2  
LONG VARCHAR.....8-3  
LOWER.....4-21  
lowercases, uppercases.....4-32  
LPAD.....4-28  
LTRIM.....4-25

**- M -**

MAKEDATE.....4-11

MAKETIME.....	4-11
mandatory column.....	8-4
MAPCHAR.....	2-10; 4-22
MAPCHAR SET.....	2-9
MAX.....	3-20; 3-21; 3-23; 4-17
MAX, character strings.....	4-34
MICROSECOND.....	4-11
MIN.....	3-20; 3-21; 3-23; 4-17
MIN, character strings.....	4-34
MINUTE.....	4-11
MOD.....	4-6
modification log.....	8-17
MONTH.....	4-11
MONTHNAME.....	4-11
multiple key.....	9-8

## - N -

named result columns.....	2-16
named result set.....	2-15; 2-16
NOROUND.....	4-7
NOT.....	3-1; 3-14; 3-16; 3-17; 3-18
not equal.....	3-1; 3-3
NOT EXCLUSIVE.....	10-3
NOT NULL.....	3-15; 8-1; 8-4
NULL.....	3-1; 5-4; 8-7
NULL value.....	2-6
NUM.....	4-21

## - O -

ON DELETE CASCADE.....	8-12
ON DELETE RESTRICT.....	8-12
ON DELETE SET DEFAULT.....	8-12
ON DELETE SET NULL.....	8-12
operator, for an outer join.....	6-6

operators, for comparisons.....	3-3
operators, for conditions.....	3-1
operators, for joins.....	6-1
optional column.....	8-4
OR.....	3-1; 3-6; 3-7; 3-18
ORDER BY.....	2-7; 3-20
order, ascending.....	2-8
order, descending.....	2-8
outer join.....	6-5

## - P -

parentheses, arithmetic.....	4-12
parentheses, set functions.....	4-20
PASSWORD.....	10-8
password, alter.....	10-8
PI.....	4-8
POWER.....	4-7
PRIMARY KEY.....	9-1; 11-10
primary key, display.....	11-10
privilege definition.....	11-14
privileges.....	11-15
privileges, create.....	10-4
privileges, grant.....	10-5
privileges, modify.....	10-7; 10-8
privileges, remove.....	10-7
PUBLIC.....	10-5

## - R -

RADIANS.....	4-9
range of values.....	3-8; 3-9; 3-10; 8-4
read lock.....	8-30
REAL.....	8-3
reference name.....	5-7
REFERENCES.....	8-11

referential integrity.....	8-11; 11-12
REFRESH.....	8-17
REPLACE.....	4-30
replacements.....	4-30
replicated tables.....	8-6
replication.....	8-6
RESOURCE.....	10-4
RESOURCE user.....	10-1
RESTRICT.....	8-12
result column names, name.....	2-16
result sets, name.....	2-15; 2-16
result tables, name.....	2-15; 2-16
REVOKE.....	10-7; 10-8
RFILL.....	4-27
ROLLBACK WORK.....	8-28
ROUND.....	4-7
row.....	2-5; 2-6; 3-1
ROWNO.....	2-6
rows, delete.....	8-10
rows, insert.....	8-7
rows, order.....	2-7
rows, search.....	9-1
rows, select.....	2-5
rows, update.....	8-9
RPAD.....	4-28
RTRIM.....	4-25

## - S -

search, character strings.....	4-33
search, using index.....	8-19
searching, special characters.....	3-13
SECOND.....	4-11
SELECT.....	2-3
SELECT *.....	2-4
SELECT DIRECT.....	9-4
SELECT DISTINCT.....	2-13

---

SELECT FIRST.....	9-6
SELECT LAST.....	9-6
SELECT NEXT.....	9-5
SELECT PREV.....	9-5
SELECT statement, general.....	2-2
SERVERDB.....	8-6
SERVERNODE.....	8-6
set functions.....	4-17
set functions, examples.....	3-19
set functions, groups.....	4-20
set functions, parentheses.....	4-20
set operations.....	7-1
short name.....	8-14
SHOW SYNONYM.....	11-9
SIGN.....	4-7
SIN.....	4-9
SINH.....	4-9
SMALLINT.....	8-3
snapshot log.....	8-17
snapshot log, drop.....	8-26
snapshot table.....	8-17
snapshot table, drop.....	8-26
sorting.....	2-7; 2-8; 4-22; 4-29; 8-7
sorting, according to word length.....	4-29
sorting, ascending.....	2-11
sorting, ASCII code.....	2-9
sorting, columns.....	2-4
sorting, descending.....	2-11
sorting, EBCDIC code.....	2-9
sorting, special characters.....	2-9
sorting, umlauts.....	2-9
SOUNDEX.....	4-22
SOUNDS.....	3-1
special characters.....	3-13
SQRT.....	4-7
STANDARD.....	10-4
STANDARD user.....	10-1
statistical information.....	11-18

STDDEV.....	4-17
string operations.....	4-21
SUBDATE.....	4-11
subqueries.....	5-1; 5-2; 5-4; 5-6
subset.....	7-6
SUBSTR.....	4-26
SUBTIME.....	4-11
subtransaction.....	8-29
SUM.....	3-21; 3-24; 4-17
SUPERDBA.....	11-12
synonym.....	8-14; 11-8
synonym definition, display.....	11-8
synonym, drop.....	8-27
SYSDBA.....	10-1
system database administrator.....	10-1
system tables.....	11-1

## - T -

table.....	2-1; 8-1; 11-2; 12-1
table definition, display.....	11-2
table, drop.....	8-26
table, replicated.....	8-6
TAN.....	4-9
TANH.....	4-9
TEMP.....	8-2
temporary tables.....	8-2
TIME.....	4-10; 4-11; 8-1; 8-3
time functions.....	4-10
TIMEDIFF.....	4-11
TIMESTAMP.....	4-11; 8-1; 8-3
transaction.....	8-28
TRANSLATE.....	4-31
trigger.....	8-23
triggers, drop.....	8-27
trigonometry.....	4-9
TRIM.....	4-25



TRUE.....	3-1
TRUNC.....	4-7; 4-28

## - U -

UNION.....	7-1
UNION ALL.....	7-5
UNIQUE.....	8-19
update.....	8-9; 8-13; 8-29
updating columns.....	8-13
UPPER.....	4-21
user classes.....	10-1
user definition, display.....	11-16
user, alter.....	10-8
user, create.....	10-3
user, drop.....	10-7
usergroup.....	10-2
usergroup, alter.....	10-8
usergroup, create.....	10-4
usergroup, drop.....	10-7

## - V -

VALUE.....	4-23
values, correlate.....	5-6
values, replace.....	4-30
VARCHAR.....	8-1; 8-2
VARIANCE.....	4-17
view.....	8-16; 9-7; 11-7
view definition, display.....	11-7
view, create.....	8-15
view, drop.....	8-26

**- W -**

WEEKOFYEAR.....	4-11
WHERE condition.....	2-5; 2-6; 3-1; 3-2; 3-3; 3-6; 3-7; 3-8; 3-9; 3-10; 3-11; .....3-12; 3-13; 3-14; 3-15; 3-16; 3-17; 3-18; 3-22; 9-7
WITH GRANT OPTION.....	10-5
WITH REPLICATION.....	8-6
write lock.....	8-30

**- Y -**

YEAR.....	4-11
-----------	------



## Table of Contents

<b>1</b>	<b>Introduction.....</b>
<b>2</b>	<b>Fundamentals.....</b>
2.1	Example of a SELECT Statement.....
2.2	Selecting Particular Columns.....
2.3	Ordering Columns.....
2.4	Selecting Particular Rows.....
2.5	.....
2.5	Ordering Rows.....
2.6	Ordering According to More than One Column.....
2.7	Eliminating Duplicate Rows.....
2.8	Keeping Result Sets.....
2.9	Introducing New Result Column Names.....
<b>3</b>	<b>Conditional Selection.....</b>
3.1	Comparison Operations.....
3.2	Multiple Conditions: AND, OR.....
3.2.1	Parentheses with AND and OR.....
3.3	Ranges of Values: BETWEEN x AND y.....
3.4	Values in a Set: IN (x,y,z).....
3.5	Searching For Character Strings.....
3.6	Negative Conditions: NOT.....
3.7	NOT with NULL, LIKE, IN, BETWEEN.....
3.8	Grouping Values: GROUP BY.....
3.9	Groups with Conditions: HAVING.....
<b>4</b>	<b>Virtual Result Columns.....</b>
4.1	Expanding the Sample.....
4.2	Arithmetic within the Result List.....
4.3	Expressions with +, -, *, / , DIV, MOD.....
4.4	Arithmetic Operations.....
4.5	Trigonometric Functions.....
4.6	Date and Time Calculations.....

4.7	Evaluation Sequence.....
4.8	Arithmetic in a Condition.....
4.9	Columns with Set Functions.....
4.10	Operations with Character Strings.....
4.10.1	Value Code Conversion.....
4.10.2	Concatenating Two CHAR Columns.....
4.10.3	Eliminating and Inserting Characters.....
4.10.4	Shortening Values.....
4.10.5	Displacing Values.....
4.10.6	Creating Bar Charts.....
4.10.7	Determining the Number of Characters.....
4.10.8	Replacements in Character Strings.....
4.10.9	Notational Checks.....
4.10.10	Searching the Position of Character Strings.....
4.10.11	Determining Minima and Maxima in Character Strings.....
<b>5</b>	<b>Subqueries.....</b>
5.1	IN, ALL, ANY, EXISTS.....
5.2	Correlation.....
<b>6</b>	<b>Columns from Two and More Tables.....</b>
6.1	Information from Two Tables.....
6.2	Information from Three Tables.....
6.3	Outer Join.....
<b>7</b>	<b>Set Operations.....</b>
7.1	UNION.....
7.2	INTERSECT.....
7.3	EXCEPT.....
<b>8</b>	<b>Data Definition and Entries.....</b>
8.1	Creating a Table.....
8.2	Data Types in a Table.....
8.3	Column Constraints.....
8.4	Replicated Tables.....
8.5	Inserting Rows.....
8.6	Updating Rows.....
8.7	Deleting Rows.....

8.8	Relations between Tables.....
8.9	Updating Column Definitions.....
8.10	Short Names for a Table.....
8.11	Creating Views.....
8.12	Creating Snapshot Tables.....
8.13	Creating an Index.....
8.14	Creating Domains.....
8.15	DB Procedures.....
8.16	Triggers.....
8.17	DB Functions.....
8.18	Dropping Objects.....
8.19	Transactions.....
<b>9</b>	<b>Accessing Single Rows.....</b>
9.1	Key Concept.....
9.2	Direct Access Using a Key.....
9.3	Position-based Access Using a Key.....
9.4	Conditional Position-based Access.....
9.5	Multiple Key.....
<b>10</b>	<b>Authorization.....</b>
10.1	User Classes.....
10.2	The SYSDBA Installs DBAs.....
10.3	The DBAs Install Additional Users.....
10.4	Modifying and Removing User Profiles.....
<b>11</b>	<b>Catalog Information.....</b>
11.1	Tables.....
11.2	Domains.....
11.3	Constraints.....
11.4	Views.....
11.5	Synonyms.....
11.6	Primary Keys.....
11.7	Indexes.....
11.8	Referential Integrity Constraints.....
11.9	Privileges.....
11.10	User Profiles.....
11.11	Administrative Information.....

11.12	Statistical Information.....
<b>12</b>	<b>The Tables Used.....</b>
<b>13</b>	<b>Index.....</b>

