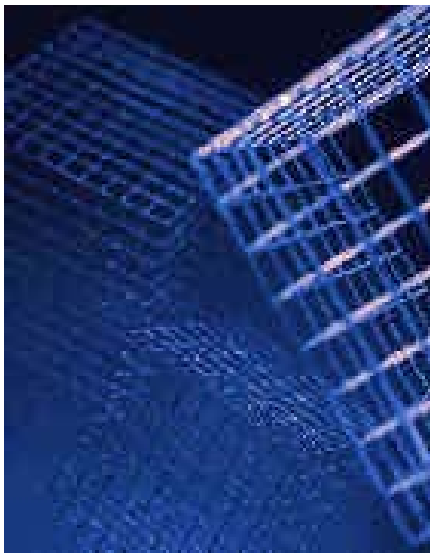
Reference Manual: SAP DB



Version 7.4

Copyright

Icons

Icon	Meaning
	Caution
	Example
	Note
	Recommendation
	Syntax

Typographic Conventions

Type Style	Description
Example text	Words or characters that appear on the screen. These include field names, screen titles, pushbuttons as well as menu names, paths and options. Cross-references to other documentation.
Example text	Emphasized words or phrases in body text, titles of graphics and tables.
EXAMPLE TEXT	Names of elements in the system. These include report names, program names, transaction codes, table names, and individual key words of a programming language, when surrounded by body text, for example, SELECT and INCLUDE.
`Example text`	Screen output. This includes file and directory names and their paths, messages, source code, names of variables and parameters as well as names of installation, upgrade and database tools.
`EXAMPLE TEXT`	Keys on the keyboard, for example, function keys (such as `F2`) or the `ENTER` key.
`Example text`	Exact user entry. These are words or characters that you enter in the system exactly as they appear in the documentation.
`<Example text>`	Variable user entry. Pointed brackets indicate that you replace these words and characters with appropriate entries.

Reference Manual: SAP DB 7.4

This document outlines the syntax and semantics of the SQL statements of the SAP DB database system, version 7.4. The syntax notation used in this document is BNF.

An SQL statement performs an operation on the database instance. The parameters used are host variables of a programming language in which the SQL statements are embedded.

For general information about the SAP DB database system, see the documentation *The SAP DB Database System* and the SAP DB homepage at *http://www.sapdb.org*.

For information on the Optimizer functions for SQL statements , see the documentation *Optimizer: SAP DB 7.4.*

Other Documentation

System Tables: SAP DB 7.4

SQL Mode ORACLE: SAP DB 7.4

Concepts

The following terms are explained here:

Privilege [Page 25]

Role [Page 25]

Database Catalog/User Data [Page 26]

Transaction [Page 210]

Subtransaction [Page 27]

Database Session [Page 27]

Data Integrity [Page 28]

Database Procedure [Page 28]

Trigger [Page 29]

SQL Mode [Page 29]

See also:

User Manual: SAP DB → Terms

Data Type

A data type is a set of values that can be represented.

- NULL value [Page 15]
- Special NULL value [Page 15]
- Non-NULL value

  Character string [Page 16], LONG column [Page 16], number [Page 16], date value [Page 17], time value [Page 17], timestamp value [Page 17], BOOLEAN [Page 17]

Use

As well as specifying the column name when you define columns [Page 118], you can also specify data types.

If required, a code attribute [Page 17] can also be entered for LONG columns and some kinds of character strings.

See also:

Using data types in SQL statements: data_type [Page 119]

NULL value

The data type [Page 15] NULL value (that is, an unspecified value) is a special value. Its relationship to any other value is always unknown.

Special NULL value

A special NULL value is a special data type [Page 15] and is the result of arithmetic operations that lead to an overflow or a division by 0.

The special NULL value is only permitted for output columns and for columns in the ORDER clause [Page 202]. If an overflow occurs in an arithmetic operation or a division by 0 at another point, the SQL statement is abnormally terminated.

The comparison of a special NULL value with any value is always undefined.

As far as sorting is concerned, the special NULL value is greater than all non-NULL values, but less than the NULL value [Page 15].

Character string

A character string is a data type [Page 15] that consists of a series of alphanumeric characters.

Examples: CHAR[ACTER] [Page 120], VARCHAR [Page 120], LONG[VARCHAR] [Page 121], DATE [Page 122], TIME [Page 123], TIMESTAMP [Page 123]

In a column definition [Page 118], a code attribute [Page 17] can be entered for the data types CHAR[ACTER], VARCHAR and LONG[VARCHAR].

The following comparison options exist for data types **CHAR[ACTER]** and **VARCHAR**:

Character strings with the same code attribute	These character strings can be compared to each other.
Character strings with the code attributes ASCII [Page 18], EBCDIC [Page 20] and UNICODE [Page 18]	These character strings are comparable with the character strings of code attributes EBCDIC, ASCII, and UNICODE, and with date [Page 17], time [Page 17] and timestamp values [Page 17].

LONG column

A LONG column is a data type [Page 15] that contains a sequence of characters of any length to which no functions can be applied.

LONG columns **cannot** be compared to one another. The contents of LONG columns **cannot** be compared to character strings [Page 16] or other data types.

See also:

LONG[VARCHAR] [Page 121]

In a column definition [Page 118], a code attribute [Page 17] can be entered for the data type LONG[VARCHAR].

Number

A number is a special data type [Page 15]. There are fixed point and floating point numbers:

- Fixed point number

  A fixed point number is described by the number of significant digits and the scale. The maximum number of significant digits is 38.

  Examples: FIXED [Page 121], INT[EGER] [Page 122], SMALLINT [Page 122]

- Floating point number

  A floating point number consists of a mantissa and an exponent. The mantissa may have up to 38 significant digits. The valid range of values for floating point numbers consists of the intervals from -9. 99999999999999999999999999999999999999E +62 to -1E-64 and from +1E-64 to +9. 99999999999999999999999999999999999999E +62 and the value 0.0.

Example: FLOAT [Page 122]

All numbers can be compared to one another.

Date value

The date value date type [Page 15] is a special character string [Page 16].

A date value can be compared to other date values and to character strings with the code attributes [Page 17] ASCII, EBCDIC, and UNICODE.

See also:

DATE [Page 122]

Date and time format [Page 50]

Time value

The time value date type [Page 15] is a special character string [Page 16].

A time value can be compared to other time values and to character strings with the code attributes [Page 17] ASCII, EBCDIC, and UNICODE.

See also:

TIME [Page 123]

Date and time format [Page 50]

Timestamp value

The timestamp value date type [Page 15] is a special character string [Page 16]. A timestamp consists of a date [Page 17] and time value [Page 17] and a microsecond specification.

A timestamp value can be compared to other timestamp values and to character strings with the code attributes [Page 17] ASCII, EBCDIC, and UNICODE.

See also:

TIMESTAMP [Page 123]

Date and time format [Page 50]

BOOLEAN

BOOLEAN is a data type [Page 15] that can only assume one of the states TRUE or FALSE and the NULL value [Page 15].

A boolean value can only be compared to other boolean values.

See also:

BOOLEAN [Page 121]

Code Attribute

For the following character strings [Page 16], a code attribute can be entered as part of a column definition [Page 118], if required: CHAR[ACTER] [Page 120], VARCHAR [Page 120], LONG[VARCHAR] [Page 121]

A code attribute defines the sort sequence to be used for comparing values.

Code Attribute	Column Values
No code attribute	Have the code attribute defined in the database system
ASCII	In ASCII code [Page 18]
BYTE	Code neutral, that is, the column values are not converted by the database system
EBCDIC	In EBCDIC code [Page 20]
UNICODE	In UNICODE [Page 18]

If you do not specify a code attribute, the code defined in the database system is used.

- The code can be defined for particular users using appropriate SQL statements (for more information, see: CREATE [Page 160]/ALTER USER Statement [Page 165] or CREATE [Page 162]/ALTER USERGROUP Statement [Page 166]).
- The code can be defined globally during the installation of the database system using the database parameter DEFAULT_CODE.

The code defined for a user overrides the code specified globally in the database system.

UNICODE

SAP DB uses the UNICODE code in line with ISO 10646, Page 1. You can find general information on UNICODE in the following documentation:

User Manual: SAP DB → *Definition of Terms* → UNICODE

User Manual: SAP DB → SAP DB as UNICODE Database

Metadata in UNICODE

The names of the database objects (such as table or column names) can be stored internally in UNICODE and can therefore be displayed in the database tools in the required presentation code.

User data in UNICODE

SAP DB supports the code attribute [Page 17] UNICODE for the data types CHAR[ACTER] [Page 120], VARCHAR [Page 120] and LONG[VARCHAR] [Page 121] and is able to map various presentation codes to the UNICODE format.

Code tables

- ASCII code [Page 18] pursuant to ISO 8859/1
- EBCDIC code [Page 20] CCSID 500, codepage 500

ASCII code

The ASCII code (in accordance with ISO 8859/1.2) is as follows:

DEC	HEX	CHAR
0	00	NUL
1	01	SOH
2	02	STX
3	03	ETX
4	04	EOT
5	05	ENQ
6	06	ACK
7	07	BEL
8	08	BS
9	09	HT
10	0A	LF
11	0B	VT
12	0C	FF
13	0D	CR
14	0E	SO
15	0F	SI
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

DEC	HEX	CHAR
32	20	SP
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

DEC	HEX	CHAR
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

DEC	HEX	CHAR
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	DEL

DEC	HEX	CHAR
128	80	
129	81	
130	82	
131	83	
132	84	
133	85	
134	86	
135	87	
136	88	
137	89	
138	8A	
139	8B	
140	8C	
141	8D	
142	8E	
143	8F	
144	90	
145	91	
146	92	
147	93	
148	94	
149	95	
150	96	
151	97	
152	98	
153	99	
154	9A	
155	9B	
156	9C	
157	9D	
158	9E	
159	9F	

DEC	HEX	CHAR
160	A0	NBSP
161	A1	¡
162	A2	¢
163	A3	£
164	A4	¤
165	A5	¥
166	A6	¦
167	A7	§
168	A8	¨
169	A9	©
170	AA	ª
171	AB	«
172	AC	
173	AD	-
174	AE	®
175	AF	¯
176	B0	°
177	B1	±
178	B2	²
179	B3	³
180	B4	´
181	B5	µ
182	B6	
183	B7	·
184	B8	¸
185	B9	¹
186	BA	º
187	BB	»
188	BC	¼
189	BD	½
190	BE	¾
191	BF	¿

DEC	HEX	CHAR
192	C0	À
193	C1	Á
194	C2	Â
195	C3	Ã
196	C4	Ä
197	C5	Å
198	C6	Æ
199	C7	Ç
200	C8	È
201	C9	É
202	CA	Ê
203	CB	Ë
204	CC	Ì
205	CD	Í
206	CE	Î
207	CF	Ï
208	D0	Ð
209	D1	Ñ
210	D2	Ò
211	D3	Ó
212	D4	Ô
213	D5	Õ
214	D6	Ö
215	D7	×
216	D8	Ø
217	D9	Ù
218	DA	Ú
219	DB	Û
220	DC	Ü
221	DD	Ý
222	DE	Þ
223	DF	ß

DEC	HEX	CHAR
224	E0	à
225	E1	á
226	E2	â
227	E3	ã
228	E4	ä
229	E5	å
230	E6	æ
231	E7	ç
232	E8	è
233	E9	é
234	EA	ê
235	EB	ë
236	EC	ì
237	ED	í
238	EE	î
239	EF	ï
240	F0	ð
241	F1	ñ
242	F2	ò
243	F3	ó
244	F4	ô
245	F5	õ
246	F6	ö
247	F7	÷
248	F8	ø
249	F9	ù
250	FA	ú
251	FB	û
252	FC	ü
253	FD	ý
254	FE	þ
255	FF	ÿ

possibly used by the operating system

EBCDIC code

EBCDIC code CCSID 500, codepage 500:

DEC	HEX	CHAR
0	00	NUL
1	01	SOH
2	02	STX
3	03	ETX
4	04	PF
5	05	HT
6	06	LC
7	07	DEL
8	08	GE
9	09	RLF
10	0A	SMM
11	0B	VT
12	0C	FF
13	0D	CR
14	0E	SO
15	0F	SI
16	10	DLE
17	11	DC1
18	12	DC2
19	13	TM
20	14	RES
21	15	NL
22	16	BS
23	17	IL
24	18	CAN
25	19	EM
26	1A	CC
27	1B	CU1
28	1C	IFS
29	1D	IGS
30	1E	IRS
31	1F	IUS

DEC	HEX	CHAR
32	20	DS
33	21	SOS
34	22	FS
35	23	
36	24	BYP
37	25	LF
38	26	ETB
39	27	ESC
40	28	
41	29	
42	2A	SM
43	2B	CU2
44	2C	
45	2D	ENQ
46	2E	ACK
47	2F	BEL
48	30	
49	31	
50	32	SYN
51	33	
52	34	PN
53	35	RS
54	36	UC
55	37	EOT
56	38	
57	39	
58	3A	
59	3B	CU3
60	3C	DC4
61	3D	NAK
62	3E	
63	3F	SUB

DEC	HEX	CHAR
64	40	SP
65	41	RSP
66	42	â
67	43	ä
68	44	à
69	45	á
70	46	ã
71	47	å
72	48	ç
73	49	ñ
74	4A	[
75	4B	.
76	4C	<
77	4D	(
78	4E	+
79	4F	!
80	50	&
81	51	é
82	52	ê
83	53	ë
84	54	è
85	55	í
86	56	î
87	57	ï
88	58	ì
89	59	ß
90	5A	]
91	5B	$
92	5C	*
93	5D	)
94	5E	;
95	5F	°

DEC	HEX	CHAR
96	60	-
97	61	/
98	62	Â
99	63	Ä
100	64	À
101	65	Á
102	66	Ã
103	67	Å
104	68	Ç
105	69	Ñ
106	6A	¦
107	6B	,
108	6C	%
109	6D	_
110	6E	>
111	6F	?
112	70	ø
113	71	É
114	72	Ê
115	73	Ë
116	74	È
117	75	Í
118	76	Î
119	77	Ï
120	78	Ì
121	79	`
122	7A	:
123	7B	#
124	7C	@
125	7D	'
126	7E	=
127	7F	"

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
128	80	Ø	160	A0	µ	192	C0	{	224	E0	\
129	81	a	161	A1	~	193	C1	A	225	E1	÷
130	82	b	162	A2	s	194	C2	B	226	E2	S
131	83	c	163	A3	t	195	C3	C	227	E3	T
132	84	d	164	A4	u	196	C4	D	228	E4	U
133	85	e	165	A5	v	197	C5	E	229	E5	V
134	86	f	166	A6	w	198	C6	F	230	E6	W
135	87	g	167	A7	x	199	C7	G	231	E7	X
136	88	h	168	A8	y	200	C8	H	232	E8	Y
137	89	i	169	A9	z	201	C9	I	233	E9	Z
138	8A	«	170	AA	¡	202	CA	-(SHY)	234	EA	²
139	8B	»	171	AB	¿	203	CB	ô	235	EB	Ô
140	8C	ð	172	AC	Ð	204	CC	ö	236	EC	Ö
141	8D	ý	173	AD	Ý	205	CD	ò	237	ED	Ò
142	8E	þ	174	AE	Þ	206	CE	ó	238	EE	Ó
143	8F	±	175	AF	®	207	CF	õ	239	EF	Õ
144	90	°	176	B0	¢	208	D0	}	240	F0	0
145	91	j	177	B1	£	209	D1	J	241	F1	1
146	92	k	178	B2	¥	210	D2	K	242	F2	2
147	93	l	179	B3	·	211	D3	L	243	F3	3
148	94	m	180	B4	©	212	D4	M	244	F4	4
149	95	n	181	B5	§	213	D5	N	245	F5	5
150	96	o	182	B6		214	D6	O	246	F6	6
151	97	p	183	B7	¼	215	D7	P	247	F7	7
152	98	q	184	B8	½	216	D8	Q	248	F8	8
153	99	r	185	B9	¾	217	D9	R	249	F9	9
154	9A	ª	186	BA		218	DA	¹	250	FA	³
155	9B	º	187	BB	|	219	DB	û	251	FB	Û
156	9C	æ	188	BC	¯	220	DC	ü	252	FC	Ü
157	9D	¸	189	BD	¨	221	DD	ù	253	FD	Ù
158	9E	Æ	190	BE	´	222	DE	ú	254	FE	Ú
159	9F	¤	191	BF	×	223	DF	ÿ	255	FF	EO

SERIAL

SERIAL is a number generator that generates positive integers starting with 1 or a specified value.

SERIAL can be used as a DEFAULT specification [Page 125] for columns (DEFAULT SERIAL) that can only contain fixed point numbers. The maximum value generated is (10**n)-1 if DEFAULT SERIAL is defined for a column of the data type FIXED [Page 121] (n).

SERIAL columns can only be assigned a value when a row is inserted. The values of a SERIAL column cannot be changed with an UPDATE statement. A SERIAL column, therefore, can be used to determine the insertion sequence and identify a row in a table uniquely.

Parameter

SQL statements for the database system can be embedded in programming languages such as C and C++. This enables the database system to be accessed from various programs. The values to be retrieved from stored in the database system can be transferred with the SQL

statements using parameters. The parameters are declared variables (so-called host variables) within the embedding program.

The data type of the host variables is defined when they are declared in the programming language. If possible, the values of the host variables are implicitly converted from the programming language data type to the data type of the database system, and vice versa.

Each parameter can be combined with an indicator variable that indicates irregularities which may have occurred when the values were assigned, for example, different value and parameter lengths, NULL value [Page 15], special NULL value [Page 15], etc. Indicator variables are essential for transferring NULL values and special NULL values. The indicator variables are declared as variables in the embedding program.

See also:

Parameter name [Page 43]

Indicator name [Page 42]

Table

- A **table** is a set of rows.
  A **row** is an ordered list of values.
  The row is the smallest unit of data that can be inserted in or deleted from a table. Each row in a table has the same number of columns [Page 24] and contains a value for each column.
- A **base table** is a table that usually has a permanent memory representation and description.
  It is also possible to create a base table that has only a temporary memory representation and description. This table and its description are implicitly dropped when a user stops working with the database system (end of session).
- A **result table** is a temporary table that is generated from one or more base table(s) by means of a SELECT statement.
- A **view table** is a table derived from base tables. A view table has a permanent description in the form of a SELECT statement.

Each table has a name that is unique within the overall database system. The names of existing tables can be used to name result tables. The original tables, however, cannot be accessed as long as the result tables exist.

If a table name was defined without an owner [Page 41], the catalog sections (part catalogs) are searched in the following order to locate the specified table name:

1. Catalog part of the current owner
2. Set of PUBLIC synonyms
3. Catalog part of the DBA who created the current user
4. Catalog part of the SYSDBA
5. Catalog part of the owner of the system tables

A table of another user can only be used if the relevant privileges have been granted.

See also:

Table name [Page 47]

Result table name [Page 41]

CREATE TABLE statement [Page 115]

CREATE VIEW statement [Page 143]

Column

All values in a table [Page 23] column have the same data type [Page 15]. A value in a column within a row is the smallest unit of data that can be modified or selected from a table or to which functions can be applied.

- An **alphanumeric column** is a character string [Page 16] column.
  All character strings in an alphanumeric column have the same length.
- A **numeric column** is either a floating point or a fixed point column.
  All numbers in a **floating point column** (floating point number [Page 16]) have the same mantissa length.
  All numbers in a **fixed point column** (fixed point number [Page 16]) have the same format; that is, the same number of digits before and after the decimal point.

Each column in a base table has a name that is unique within the table.

See also:

Column name [Page 46]

Using column definitions in SQL statements: column_definition [Page 118]

Domain

Domain definitions enable ranges of values to be defined and designated for table columns [Page 24].

Each value range definition has a name that is unique within the overall database system.

If a domain was defined without an owner [Page 41], the catalog sections (part catalogs) are searched in the following order to locate the specified value range:

1. Catalog part of the current owner
2. Catalog part of the DBA who created the current user
3. Catalog part of the SYSDBA

See also:

Domain name [Page 41]

CREATE DOMAIN statement [Page 140]

Index

Indexes speed up access to rows in a table. They can be created for a single column or for a series of columns. When defining indexes, you specify whether the indexed column values in the different rows must be unique or not.

The assigned index name [Page 42] and table name [Page 47] must be unique.

See also:

CREATE INDEX statement [Page 148]

Synonym

A synonym is another name for a table [Page 23].

Every synonym has a name that is unique within the entire database system and differs from all the other table names.

See also:

Synonym name [Page 46]

CREATE SYNONYM statement [Page 142]

Users and Usergroups

The following user names and passwords are defined when the database system is installed.

- DBM operator
- Database system administrator (SYSDBA)
- DOMAIN user

There are four database user classes in WARM database mode:

- Database system administrator (SYSDBA)
- Database administrators (DBA users)
- RESOURCE users
- STANDARD users

Usergroups can also be defined. All of the members of a usergroup have the same rights with regard to data assigned to the group.

See also:

User Manual: SAP DB → User Concept

User Manual: SAP DB → Definition of Terms

User name [Page 40]

Usergroup name [Page 39]

CREATE USER statement [Page 160]

CREATE USERGROUP statement [Page 162]

Privilege

A privilege is used to impose restrictions on operations carried out on certain objects.

Users can only execute operations on objects if they have been granted the privileges to do so. The owner of an object receives all of the relevant privileges when the object is created. Privileges can be explicitly granted to other users. Privileges are not granted to other users implicitly.

Users who are not the owner of an object can only grant privileges to other users if they have already been granted these privileges and are allowed to pass them on, i.e. with the relevant option.

See also:

Privilege type [Page 43]

GRANT statement [Page 170]

Role

A role is a collection of privileges [Page 25].

Like a privilege, a role can be assigned to a different role or to a user.

While privileges are always valid, roles are always inactive, that is, the privileges they contain are not valid. Roles can be activated for individual database sessions. Every user to which roles were assigned can also define which of the roles should be active in each of his or her

database sessions. This definition can be changed after a database session has been opened.

All roles are inactive for the current database session while data definition commands are being executed.

See also:

Role name [Page 45]

CREATE ROLE statement [Page 169]

DROP ROLE statement [Page 170]

Role concept

Database Catalog/Application Data

Logical data storage takes place in the following areas of a SAP DB database:

- The **database catalog** comprises metadata containing the definitions of database objects such as base tables [Page 23], view tables [Page 23], synonyms [Page 24], domains [Page 24], indexes [Page 24], and users and usergroups [Page 25].
  See also: *User Manual: SAP DB*, Database Catalog section.
- **Application Data**: *User Manual: SAP DB*, Application Data section**.**

Transaction

A transaction is a sequence of SQL statements that are handled by the database system as a basic unit, in the sense that any modifications made to the database by the SQL statements are either all reflected in the state of the database, or else none of the database modifications are retained.

The first transaction is opened when a database session [Page 27] is opened with the CONNECT statement [Page 211]. The transaction is concluded with the COMMIT statement [Page 213] or the ROLLBACK statement [Page 213]. When a transaction is successfully concluded with a COMMIT statement, all of the changes to the database are retained. If a transaction is aborted using a ROLLBACK statement, on the other hand, or if it is aborted in another way, all of the changes to the database made by the transaction are rolled back.

Both the COMMIT and ROLLBACK statements open a new transaction implicitly.

A transaction can be divided into other basic units, subtransactions [Page 27].

Locks

Since the database system permits concurrent transactions on the same database objects, locks on rows, tables, and the database catalog are necessary to isolate individual transactions.

For information about the lock concept, see the *User Manual: SAP DB*, Lock Behavior section.

- The assignment of implicit locks can be affected by the setting of the isolation level with the CONNECT statement [Page 211].
- Locks can be assigned explicitly using the LOCK statement [Page 215] or by the assignment of a LOCK option [Page 203].
- Exclusive locks for rows that have not yet been modified, and share locks on rows can be released by the UNLOCK statement [Page 216] before the end of the transaction.

The locks assigned to a transaction are usually released at the end of the transaction, making the respective database objects accessible again to other transactions.

SQL statements for transaction management

CONNECT statement [Page 211]	SET statement [Page 212]	
COMMIT statement [Page 213]	ROLLBACK statement [Page 213]	SUBTRANS statement [Page 214]
LOCK statement [Page 215]	UNLOCK statement [Page 216]	RELEASE statement [Page 217]

Subtransaction

The purpose of closed, nested transactions [Page 210] (subtransactions) is to let a series of database operations within a transaction appear as a unit with regard to modifications to the database.

Subtransactions are preceded by SUBTRANS BEGIN and closed by SUBTRANS END or SUBTRANS ROLLBACK.

- If a subtransaction is concluded with SUBTRANS END, any modifications made are kept.
- If a subtransaction is closed with SUBTRANS ROLLBACK, all modifications made to the database system are reversed. Modifications made by subtransactions contained in this subtransaction are also reversed, even if they were concluded with SUBTRANS END.

SUBTRANS END and SUBTRANS ROLLBACK do not affect locks. These are only released by COMMIT or ROLLBACK. COMMIT or ROLLBACK implicitly close all subtransactions.

See also:

SUBTRANS statement [Page 214]

COMMIT statement [Page 213]

ROLLBACK statement [Page 213]

Database Session

You will find an explanation of the term in *User Manual: SAP DB → Definition of Terms →* Database session.

See also:

Users and user groups [Page 25]

Password [Page 42]

CONNECT statement [Page 211]

SET statement [Page 212]

Data integrity

- **Integrity rules**: the database system provides a wide range of declarative integrity rules, thus reducing the programming requirements for applications. Integrity rules that refer to a table can also be specified (see constraint name [Page 40]).
- **Key**: a key comprising one or more columns can be defined for each table. The database system ensures that each key is unique. A key can also consist of columns of different data types (see key definition [Page 131]).
- **UNIQUE definition**: the uniqueness of the values in other columns and column combinations can also be ensured by using other mechanisms (see UNIQUE definition [Page 132] for "alternate keys").
- **NOT NULL**: by specifying NOT NULL, you can ensure that the NULL value is not accepted in individual columns.
- **DEFAULT definition**: you can define default values for each column (see DEFAULT specification [Page 125]).
- **Referential integrity conditions**: by specifying referential integrity conditions, you can declare deletion and existence dependencies between the rows in two tables (see name of a referential constraint [Page 44]).
- **Database procedures and triggers**: complex integrity rules that require access to further tables can be formulated with database procedures [Page 28] or triggers [Page 29].

Database procedure

In a well-structured database application, SQL statements are typically not distributed over the entire application but concentrated in a single access layer instead. This access layer has a procedural interface to the rest of the application at which the operations for application objects are made available in form of abstract data types.

In client/server configurations, the client and server interact when an SQL statement is executed in the access layer. The number of these interactions can be reduced considerably by transferring the SQL access layer from the client to the server.

SAP DB provides a language (special SQL syntax) for this purpose that allows an SQL access layer to be formulated on the server side. This special SQL syntax can be used to define database procedures and triggers [Page 29].

This has three main advantages:

- The number of interactions between client and server is reduced considerably (several factors). Client/server communication is only required for each operation on the application object, and not for each SQL statement. This enhances the performance of client-server configurations considerably.
- The SQL access layer contains the procedurally formulated integrity and business rules. By concentrating these rules on the server side and eliminating them from the database applications, modifications can be made centrally and thus become valid immediately in all database applications. In this way, the integrity and decision rules also become a part of the catalog in the database system.
- An SQL access layer in the form of database procedures transferred to the server side is an essential customizing tool, as it allows customer-specific database functionality to be included.

To be able to execute a database procedure, users must have the call privilege. This call privilege is independent of the user privileges for the tables and columns used in the database procedure. As a result, users may be able to use a database procedure to execute SQL statements that they otherwise would not have access to.

Database procedures are called explicitly from the programming language of the application. They can contain parameters, except for LONG column [Page 16]s. The extent to which LONG columns can be used within database procedures depends on the length of the LONG columns and the amount of storage space available.

As with any SQL statement, precautions must be taken to ensure that calling a database procedure has the desired effect, and that errors do not have any lasting effects on the database system. SAP DB provides nested transactions for this purpose. Each database procedure call can run in a subtransaction [Page 27] that can be reset without interfering with transaction control in the database application.

See also:

Name of a database procedure [Page 40]

Trigger

While database procedures [Page 28] are called explicitly from the programming language of the application, triggers are special procedures that run implicitly on a base table (or a view table built on this base table) after a data manipulation statement has been executed.

The conditions under which a trigger is to be executed can be restricted further.

The trigger is executed for each row to which the SQL statement refers. The trigger can access both the old values (values before update or deletion) and the new values (values after update or insertion) in this row.

A trigger can call further triggers implicitly.

Triggers can be used to check complicated integrity rules, to initiate derived database modifications for the row in question, or to implement complex access protection rules.

SAP DB provides a language (special SQL syntax) that can be used to define database procedures and triggers.

See also:

Trigger name [Page 47]

SQL mode

You will find an explanation of the term in *User Manual: SAP DB → Definition of Terms →* SQL mode.

This document describes the functionality of the database system provided by the INTERNAL SQL mode.

Only those SQL statements are described for which the same SQL mode is used to generate the object and execute the statement for the object. If database objects are created in one SQL mode and addressed in another, the object may have properties that are not known in the current SQL mode and, therefore, cannot be described.

SQL statement for specifying the SQL mode

CONNECT statement [Page 211]

Basic Elements

Character [Page 30]

Literal [Page 31]

Character

A `character` is an element of a character string [Page 16] or keyword [Page 35].

Syntax

```
<character> ::= <digit>
              | <letter>
              | <extended_letter>
              | <hex_digit>
              | <language_specific_character>
              | <special_character>
```

digit [Page 30], letter [Page 30], extended_letter [Page 31], hex_digit [Page 31], language_specific_character [Page 31], special_character [Page 31]

Digit

A `digit` is a character [Page 30].

Syntax

```
<digit> ::= 0 | 1 | 2 | | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Letter

A `letter` is a character [Page 30].

Syntax

```
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N |
O | P | Q | R | S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |
r | s | t | u | v | w | x | y | z
```

Extended letter

An extended letter is a character [Page 30].

Syntax

```
<extended_letter> ::= # | @ | $
```

hex_digit

A `hex_digit` is a character [Page 30].

Syntax

```
<hex digit> ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| A | B | C | D | E | F
| a | b | c | d | e | f
```

language_specific_character

A language-specific character [Page 30] is any letter that occurs in a northern, southern, or central European language and is not contained in the list of letters [Page 30].

German umlauts: ä, ö, ü

French letters with a “grave” accent.

If you have installed a UNICODE [Page 18]-enabled database, a language-specific character is a character that is not included in the ASCII-Code [Page 18] list from 0 to 127.

Special character

A `special character` is any character [Page 30] that is not contained in the following list:

- digit [Page 30]
- letter [Page 30]
- extended_letter [Page 31]
- hex_digit [Page 31]
- language_specific_character [Page 31]
- Characters that indicate the end of a line in a file

Literal (literal)

A `literal` is an unknown data object that is defined fully by virtue of its value (specifies a non-NULL value, see data type [Page 15]). Literal values cannot be modified. A distinction is made between string literals and numeric literals.

Syntax

```
<literal> ::= <string literal> | <numeric literal>
```

string_literal [Page 32], numeric_literal [Page 33]

String literals
```
'69190 Walldorf'
'Anthony Smith'
```

Numeric literals
```
+0.58498
1E160
-765E-04
```

String Literal (string_literal)

A string literal [Page 31] is a sequence of characters in quotation marks. String literals can also be represented in hexadecimal notation by preceding them with x or X.

Syntax

```
<string_literal> ::= '' | '<character>...' | <hex_literal>
```

Character [Page 30], hex_literal [Page 32]

```
'69190 Walldorf'
```

```
'Anthony Smith'
```

```
X'12ab'
```

Explanation

A quotation mark within a character string is represented by two successive quotation marks.

A string literal of the type `'<character>...'` or `''` is only valid for a value referring to an alphanumeric column with the code attribute [Page 17] ASCII or EBCDIC. A hex literal [Page 32] is only valid for a value referring to a column with the code attribute BYTE.

A string literal of the type `''`, `x''` and `X''`, and string literals that only contain blanks are **not** the same as the NULL value [Page 15].

hex_literal

String literal [Page 32] that contains a value in hexadecimal notation.

Syntax

```
<hex_literal> ::= x'' | X'' | x'<hex_digit_seq>' | X'<hex_digit_seq>'
```

hex_digit_seq [Page 32]

```
x'123F'
X'12ab'
```

hex_digit_seq

Sequence of hexadecimal digits (`hex_digit_seq`).

Syntax

```
<hex_digit_seq> ::=
<hex_digit><hex_digit> | <hex_digit_seq><hex_digit><hex_digit>
```

hex_digit [Page 31]

Numeric Literal (numeric_literal)

A numeric literal [Page 31] is a number [Page 16] represented as a fixed or floating point number.

Syntax

```
<numeric_literal> ::= <fixed_point_literal> |
<floating_point_literal>
```

fixed_point_literal [Page 33], floating_point_literal [Page 33]

Fixed point literal

Numeric literal [Page 33] that specifies a number [Page 16] as a fixed point number.

Syntax

```
<fixed_point_literal> ::=
  [<sign>]<digit_sequence>[.<digit_sequence>]
| [sign]<digit_sequence>. | [sign].<digit_sequence>
```

sign [Page 33], digit_sequence [Page 33]

```
+123.12
1234.
```

Sign

Sign

Syntax

```
<sign> ::= + | -
```

Digit sequence

Sequence of digits

Syntax

```
<digit_sequence> ::= <digit>...
```

digit [Page 30]

Floating point literal

Numeric literal [Page 33] that specifies a number [Page 16] as a floating point number.

Syntax

```
<floating_point_literal> ::=  <mantissa>E<exponent> |
<mantissa>e<exponent>
```

mantissa [Page 34], exponent [Page 34]

```
1e160
-765E-04
```

Mantissa

Mantissa

Syntax

```
<mantissa> ::= <fixed_point_literal>
```

fixed_point_literal [Page 33]

Exponent

Exponent

Syntax

```
<exponent> ::= [<sign>][[<digit>]<digit>]<digit>
```

sign [Page 33], digit [Page 30]

Unsigned integer

An `unsigned_integer` is a special numeric literal.

Syntax

```
<unsigned_integer> ::= <numeric_literal>
```

numeric_literal [Page 33]

Explanation

An unsigned integer can be represented in any way but must be a positive integer.

Integer

An integer is a special numeric_literal [Page 33]. This integer can be displayed in any number of ways.

Syntax

```
<integer> ::= [sign]<unsigned_integer>
```

sign [Page 33], unsigned_integer [Page 34]

Token

A character set or `token` comprises a series of characters that are combined to form a lexical unit. A distinction is made between regular and delimiter tokens.

Syntax

```
<token> ::= <regular token> | <delimiter token>
```

regular_token [Page 35], delimiter_token [Page 38]

```
SELECT * FROM reservation
ALTER TABLE reservation DROP FOREIGN KEY
customer_reservation
```

Explanation

Each token can be followed by any number of blanks. Each `regular_token` must be followed by a `delimiter_token` or a blank.

Double quotes [Page 37] within a special identifier [Page 37] are represented by two consecutive quotes.

Regular token

Normal character set (token [Page 35]) (`regular_token`)

Syntax

```
<regular_token> ::= <literal> | <keyword> | <identifier> |
<parameter_name>
```

Literal [Page 31], key_word [Page 35], identifier [Page 36], parameter_name [Page 43]

```
SELECT
'Tours10'
```

Keyword

Keyword. A distinction is made between „normal“ and reserved keywords.

Syntax

```
<keyword> ::= <not_reserved_key_word> | <reserved_keyword>
```

Not_reserved_keyword [Page 36], reserved_keyword [Page 36]

Explanation

Keywords can be entered in uppercase/lowercase characters.

Reserved keywords must not be used in simple identifiers [Page 36]. Reserved keywords, however, can be specified in the form of special identifiers [Page 37].

Not reserved keyword

Keywords [Page 35] (`not reserved keywords`). If possible, these key words should not be used to designate objects.

You can find a list of all keywords in the syntax directory: not_reserved_key_word

Reserved keyword

Reserved keywords [Page 35] (`reserved key_word`). These key words must not be used to designate objects.

You can find a list of all keywords in the syntax directory: reserved_key_word

Identifier

Identifier (`identifier`). A distinction is made between simple identifiers and special identifiers.

Syntax

```
<identifier> ::= <simple identifier> | <double quotes><special
identifier><double quotes>
```

simple_identifier [Page 36], double_quotes [Page 37], special_identifier [Page 37]

Explanation

Identifiers can be entered in uppercase/lowercase characters. When you specify simple identifiers, upper and lower case are ignored, as the system always converts the identifier to upper case letters.

Reserved keywords [Page 36] must not be used in simple identifiers. Reserved keywords, however, can be specified in the form of special identifiers.

Double quotation marks within a special identifier are represented by two consecutive quotation marks.

Simple identifier: `reservation`

Special identifier: `"ADD"`

Simple Identifier (simple_identifier)

Simple identifier [Page 36]. The first character in a simple identifier must not be a digit or underscore [Page 37].

Syntax

```
<simple_identifier> ::=
<first_character>[<identifier_tail_character>...]
```

first_character [Page 37], identifier_tail_character [Page 37]

Explanation

Simple identifiers are always converted into uppercase characters in the database. For this reason, simple identifiers are not case sensitive.

Reserved keywords [Page 36] must not be used in simple identifiers.

If the name of a database object is to contain lowercase letters, special characters, reserved keywords, or blanks, the identifier must be specified as a special identifier [Page 37] (enclosed in double quotation marks).

First character

First character in a simple identifier [Page 36].

Syntax

```
<first_character> ::= <letter> | <extended_letter> |
<language_specific_character>
```

letter [Page 30], extended_letter [Page 31], language_specific_character [Page 31]

Identifier tail character

Character allowed after the first character in a simple identifier [Page 36] or password [Page 38].

Syntax

```
<identifier_tail_character> ::=
<letter> | <extended_letter> | <language_specific_character>
| <digit> | <underscore>
```

letter [Page 30], extended_letter [Page 31], language_specific_character [Page 31], digit [Page 30], underscore [Page 37]

Underscore

Underscore

Syntax

```
<underscore> ::= _
```

Double quotes

Quotation marks (`double_quotes`)

Syntax

```
<double_quotes> ::= "
```

Special Identifier (special_identifier)

Special identifier (`special_identifier`). A special identifier must be entered in **double quotation marks** if it is to be used as an identifier [Page 36].

Syntax

```
<special_identifier> ::= <special_identifier_character>...
```

`<special_identifier_character> ::=` any characters [Page 30], that can be linked in any sequence

Explanation

Special identifiers are always used as specified in the database; that is upper and lower case characters are taken into account. Special identifiers are case-sensitive.

If the name of a database object is to contain lowercase letters, special characters, reserved keywords (reserved_key_word [Page 36]), or blanks, the identifier must be specified as a special identifier (enclosed in double quotation marks).

“ADD”, “Example_1”

Delimiter token

Delimiter token [Page 35]

Syntax

```
<delimiter_token> ::=
   ( | ) | , | . | + | - | * | / | < | > | <> | != | = | <= | >=
| ¬= | ¬< | ¬>  (for machines with EBCDIC code [Page 20])
| ~= | ~< | ~>  (for machines with ASCII code [Page 18])
```

Names

Names identify objects. The following list contains names that are frequently used in the syntax description of the SQL statements.

Name	
Alias name [Page 39]	`alias_name`
Column name [Page 46]	`column_name`
Constraint name [Page 40]	`constraint_name`
Name of a database procedure [Page 40]	`dbproc_name`
Domain name [Page 41]	`domain_name`
Index name [Page 42]	`index_name`
Indicator name [Page 42]	`indicator_name`
Mapchar set name [Page 42]	`mapchar_set_name`
Owner [Page 41]	`owner`
Parameter name [Page 43]	`parameter_name`
Password [Page 42]	`password`
Reference name [Page 45]	`reference_name`
Name of a referential constraint [Page 44]	`referential_constraint_name`
Result table name [Page 41]	`result_table_name`

Role name [Page 45]	role_name
Sequence name [Page 45]	sequence_name
Synonym name [Page 46]	synonym_name
Table name [Page 47]	table_name
Trigger name [Page 47]	trigger_name
Usergroup name [Page 39]	usergroup_name
User name [Page 40]	user_name

Explanation

For all names consisting of several identifiers that are separated by a ".", any number of blanks can be specified before and after the dot.

Alias name

An alias name is a new column name [Page 46] that specifies the name of a column in the following types of tables:

- View tables
- Tables defined with a recursive DECLARE CURSOR statement

Syntax

```
<alias_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for defining an alias name

CREATE VIEW statement [Page 143]

Recursive DECLARE CURSOR statement [Page 186]

Usergroup name

A usergroup name identifies a usergroup [Page 25].

Syntax

```
<usergroup_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for defining a usergroup name

CREATE USERGROUP statement [Page 165]

User name

A user name defines a user [Page 25].

Syntax

```
<user_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for defining a user name

CREATE USER statement [Page 160]

Constraint name

A constraint name defines a condition (data integrity [Page 28]) that must be satisfied by the rows in a table.

Syntax

```
<constraint_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statements for defining a constraint name

CONSTRAINT definition [Page 127]

CREATE TABLE statement [Page 115]

ALTER TABLE statement [Page 133]

Name of a database procedure (dbproc_name)

The name of a database procedure (`dbproc_name`) designates a database procedure [Page 28].

Syntax

```
<dbproc_name> ::= [<owner>.]<procedure_name>
<procedure_name> ::= <identifier>
```

owner [Page 41], identifier [Page 36]

Explanation

You cannot specify TEMP as the owner in a database procedure.

SQL statements for creating, calling, and dropping a database procedure

CREATE DBPROC statement [Page 152]

CALL statement [Page 183]

DROP DBPROC statement [Page 158]

Domain name

A domain name identifies a domain [Page 24] in a table column [Page 24].

Syntax

```
<domain_name> ::= [<owner>.]<identifier>
```

owner [Page 41], identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for defining a domain name

CREATE DOMAIN statement [Page 140]

You cannot specify TEMP as the owner in a domain name.

Owner

The owner of an object is defined by specifying the name of owner name.

Syntax

```
<owner> ::= <user_name> | <usergroup_name> | TEMP
```

user_name [Page 40], usergroup_name [Page 39]

Explanation

If the owner is not a member of a usergroup, the owner name and usergroup name are identical.

If TEMP is specified as the owner in a table name [Page 47], the table is a temporary table. The owner of this temporary table is the current owner.

An error message is output if the name of the owner has more than 32 characters.

Result table name

A result table name identifies a result table (see table [Page 23]).

Syntax

```
<result_table_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for defining a result table name

QUERY statement [Page 184]

Index name

An index name identifies an index [Page 24].

Syntax

```
<index_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for creating an index

CREATE INDEX statement [Page 148]

Indicator name

An indicator name designates an indicator variable in an application that can be specified together with a parameter name.

Syntax

```
<indicator_name> ::= <parameter_name>
```

parameter_name [Page 43]

Explanation

The indicator variable of a parameter [Page 22] provides information on any irregularities (e.g. NULL value, difference value and parameter lengths).

Mapchar Set Name (mapchar_set_name)

A Mapchar set name (`mapchar_set_name`) identifies a Mapchar set.

Syntax

```
<mapchar_set_name> ::= <identifier>
```

identifier [Page 36]

Function in which Mapchar sets are used

MAPCHAR [Page 86]

See also:

User Manual: SAP DB, Language Support (Mapchar Set) section

Password

Users require a password to connect to the database instance (start a database session [Page 27]).

Syntax

```
<password> ::= <identifier> |
<first_password_character>[<identifier_tail_character>...]
```

identifier [Page 36], identifier_tail_character [Page 37]

```
<first_password_character> ::= <letter> | <extended_letter> |
<language_specific_letter> | <digit>
```

letter [Page 30], extended_letter [Page 31], language_specific_letter [Page 31], digit [Page 30]

Explanation

Passwords are truncated after 18 characters.

SQL statement for defining a user password

CREATE USER statement [Page 160]

SQL statement for changing a user password

ALTER PASSWORD statement [Page 169]

Parameter name

A parameter name identifies a parameter [Page 22] (host variable) in an application containing SQL statements from the database system.

Syntax

```
<parameter_name> ::= :<identifier> | :<identifier>(<identifier>) |
:<identifier>(.<identifier>.)
```

identifier [Page 36]

Explanation

The conventions of the programming language in which the SQL statements of the database system are embedded determine the number of significant characters in the parameter name.

Identifiers for parameter names may contain the characters "." and "_", but not as the first character.

Privilege type (privilege)

A privilege type (`privilege`) identifies a certain privilege [Page 25].

Syntax

```
<privilege> ::= INSERT | UPDATE [(<column_name>,...)]
| SELECT [(<column_name>,...)] | SELUPD [(<column_name>,...)]
| DELETE | INDEX | ALTER | REFERENCES [(<column_name>,...)]
```

column_name [Page 46]

Explanation

Privilege	Explanation
INSERT	Allows the identified user to insert rows in the specified table. The current user must be authorized to grant the INSERT privilege.

UPDATE	Allows the identified user to update rows in the specified table. If column names are specified, the rows may only be updated in the columns identified by these names. The current user must be authorized to grant the UPDATE privilege.
SELECT	Allows the identified user to select rows in the specified table. If column names are specified, the rows may only be selected in the columns identified by these names. The current user must be authorized to grant the SELECT privilege.
SELUPD	The SELECT and UPDATE privileges are granted. If column names are specified, the rows may only be selected or updated in the columns identified by these names. The current user must be authorized to grant both the SELECT and the UPDATE privileges.
DELETE	Allows the identified user to delete rows from the specified table. The current user must be authorized to grant the DELETE privilege.
INDEX	Allows the identified user to execute the CREATE INDEX and DROP INDEX statements for the specified tables. The INDEX privilege can only be granted for base tables. The current user must be authorized to grant the INDEX privilege.
ALTER	Allows the identified user to execute the ALTER TABLE statement for the specified tables. The ALTER privilege can only be granted for base tables. The current user must be authorized to grant the ALTER privilege.
REFERENCES	Allows the identified user to specify the table as a referenced table in a column_definition [Page 118] or referential_constraint_definition [Page 128].

SQL statement for granting or revoking privileges

GRANT statement [Page 170]

REVOKE statement [Page 172]

Name of a referential constraint (referential_constraint_name)

The name of a referential constraint (`referential_constraint_name`) identifies a referential constraint (referential integrity condition, data integrity [Page 28]). This integrity condition specifies deletion and existence dependencies between two tables.

Syntax

```
<referential_constraint_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for defining a referential constraint

Referential CONSTRAINT definition [Page 128]

CREATE TABLE statement [Page 115]

ALTER TABLE statement [Page 133]

Reference name

The reference name is an identifier that is declared for a certain validity range and associated with exactly one table [Page 23]. The scope of this declaration is the entire SQL statement.

Syntax

```
<reference_name> ::= <identifier>
```

identifier [Page 36]

Explanation

The same reference name specified in various scopes can be associated with different tables or with the same table.

An error message is output if the name has more than 32 characters.

Role Name (role_name)

A role name identifies a role [Page 25].

Syntax

```
<role_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statement for defining a role

CREATE ROLE statement [Page 169]

SQL statement to assign privileges to a role

GRANT statement [Page 170]

SQL statement for granting a role

GRANT statement [Page 170]

SQL statements for activating a role

ALTER USER statement [Page 165]

ALTER USERGROUP statement [Page 166]

SET statement [Page 212]

SQL statement for dropping a role

DROP ROLE statement [Page 170]

See also:

Role concept

Sequence name

A sequence name identifies a sequence of values.

Syntax

```
<sequence_name> ::= <identifier>
```

identifier [Page 36]

Explanation

A sequence is a series of values that are generated in accordance with certain rules. The step width between these values can be defined, among other things.

Sequences can be used to generate unique values. These are not uninterrupted, because values generated within a transaction that was rolled back cannot be used again.

An error message is output if the name has more than 32 characters.

SQL statement for defining a sequence name

CREATE SEQUENCE statement [Page 140]

Column name

A column name identifies a column [Page 24].

Syntax

```
<column_name> ::= <identifier>
```

identifier [Page 36]

Explanation

An error message is output if the name has more than 32 characters.

SQL statements for defining a column name

CREATE TABLE statement [Page 115]

CREATE VIEW statement [Page 143]

ALTER TABLE statement [Page 133]

QUERY statement [Page 184]

Synonym name

A synonym name identifies a synonym [Page 24] for a table name.

Syntax

```
<synonym_name> ::= <identifier>
```

identifier [Page 36]

Explanation

If the synonym is not a PUBLIC synonym, it is only known by one user or usergroup [Page 25]. A PUBLIC synonym is known by every user and user group.

An error message is output if the name has more than 32 characters.

SQL statement for defining a synonym name

CREATE SYNONYM statement [Page 142]

Table name

A table name identifies a table [Page 23].

Syntax

```
<table_name> ::= [<owner>.]<identifier>
```

owner [Page 41], identifier [Page 36]

Explanation

The database system uses certain table names for internal purposes. The identifiers [Page 36] of these tables begin with SYS. To avoid naming conflicts, therefore, you should not use table names that start with SYS.

If the name of the owner [Page 41] was not specified in the table name, the catalog parts are searched for the table name in the following order:

1. Catalog part of the current owner
2. Set of PUBLIC synonyms
3. Catalog part of the DBA who created the current user
4. Catalog part of the SYSDBA
5. Catalog part of the owner of the system tables

An error message is output if the name has more than 32 characters.

SQL statements for defining a table name

CREATE TABLE statement [Page 115]

CREATE VIEW statement [Page 143]

CREATE SYNONYM statement [Page 142]

Trigger Name (trigger_name)

A trigger name identifies a trigger [Page 29] that is defined for a table.

Syntax

```
<trigger_name> ::= <identifier>
```

identifier [Page 36]

Explanation

A trigger name must not exceed 32 characters in length.

SQL statements for creating and dropping a trigger

CREATE TRIGGER statement [Page 158]

DROP TRIGGER statement [Page 159]

Column specification (column_spec)

A column specification (`column_spec`) specifies a column in a table.

Syntax

```
<column_spec> ::= <column_name> | <table_name>.<column_name>
| <reference_name>.<column_name> | <result_table name>.<column_name>
```

column_name [Page 46], table_name [Page 47], reference_name [Page 45], result_table_name [Page 41]

Explanation

For all names consisting of several identifiers [Page 36] separated by a ".", any number of blanks can be specified before and after the dot.

Parameter specification (parameter spec)

A parameter specification (`parameter spec`) identifies the parameter name and, if necessary, the indicator name that are necessary to specify a parameter.

Syntax

```
<parameter spec> ::= <parameter name> [<indicator name>]
```

parameter_name [Page 43], indicator name [Page 42]

Explanation

The indicator must be declared as a variable in the embedding programming language. It must be possible to assign at least four-digit integers to this variable.

A distinction is made between output parameters and input parameters:

- **Output parameters**: parameters that are to receive values retrieved from the database system.
  - An indicator parameter with the value 0 indicates that the transferred value is not a NULL value and that the parameter value is the transferred value.
  - An indicator with the value –1 indicates that the parameter value is the NULL value.
  - Alphanumeric output parameters: an indicator with a value greater than 0 indicates that the assigned character string was too long and was truncated as a result. The indicator indicates the untruncated length of the original output column.
  - Numeric output parameters: an indicator with a value greater than 0 indicates that the assigned value has too many significant digits and decimal places have been truncated. The indicator indicates the number of digits in the original value.
  - With numeric output parameters, an indicator with the value -2 indicates that the parameter value is the special NULL value [Page 15].
- **Input parameters**: parameters containing values that are to be transferred to the database system.
  - An indicator with a value greater than or equal to 0 indicates that the specified parameter value is the value that is to be transferred to the database system.
  - An indicator with the value less than 0 indicates that the specified parameter value is the NULL value.

Specifying values (extended value spec)

Values can be specified (`extended value spec`) by specifying values (`value spec`) or with one of the keywords DEFAULT or STAMP.

Syntax

```
<extended value spec> ::= <value spec> | DEFAULT | STAMP
```

value_spec [Page 49]

Explanation

- DEFAULT keyword
  DEFAULT identifies the default value for the column in a CREATE TABLE statement [Page 115] or ALTER TABLE statement [Page 133]. DEFAULT cannot be used to specify values if one of these values is not defined.
  The DEFAULT keyword can be used in the following **SQL statements**:
  INSERT statement [Page 173]
  UPDATE statement [Page 178]
  The DEFAULT keyword can be used in a DEFAULT predicate [Page 60].
- STAMP key word
  The database system is able to generate unique values. This is a serial number that starts with X'000000000001'. The values are assigned in ascending order. It cannot be ensured that a sequence of values is uninterrupted. The STAMP key word supplies the next value generated by the database system.
  The STAMP keyword can be used in the following **SQL statements** (only on columns of the data type CHAR(n) BYTE with n>=8, see DEFAULT specification [Page 125]):
  INSERT statement [Page 173]
  UPDATE statement [Page 178]
  If the user wants to find out the generated value before it is applied to the column, the following **SQL statement** must be used:
  NEXT STAMP statement [Page 183]

Specifying Values (value_spec)

Values can be specified (`value_spec`) by specifying literals, parameter specifications, or a series of keywords.

Syntax

```
<value_spec> ::= <literal> | <parameter_spec>
| NULL | USER | USERGROUP | SYSDBA | UID
| [<owner>.]<sequence_name>.NEXTVAL |
[<owner>.]<sequence_name>.CURRVAL
| <table_name>.CURRVAL
| DATE | TIME | TIMESTAMP | UTCDATE | TIMEZONE | UTCDIFF
| TRUE | FALSE | TRANSACTION
```

`literal`	Literal [Page 31]
`parameter_spec`	Parameter specification [Page 48]
`NULL`	NULL value [Page 15]
`USER`	Current user name [Page 40]
`USERGROUP`	Name of the user group [Page 39] to which the user calling the SQL statement belongs. If the user does not belong to a user group, the user name is displayed.

SYSDBA	SYSDBA of the database instance
UID	Identification of the current user. This is a whole number.
[<owner>.]<sequence_name>.NEXTVAL	Next value generated for the specified sequence name [Page 45] (of the owner [Page 41] in question).
[<owner>.]<sequence_name>.CURRVAL	Value that was generated using [<owner>.]<sequence_name>.NEXTVAL as the final value for the specified sequence name.
<table_name>.CURRVAL	Last assigned value of the serial column in the table_name [Page 47] table in the current database session.
DATE	Current date [Page 17]
TIME	Current time [Page 17]
TIMESTAMP	Current timestamp [Page 17]
UTCDATE	Current UTC time stamp (Greenwich Mean Time)
TIMEZONE	Time difference in hours in the format hhmmss (in data type FIXED(6)) between your local time value and the UTC time value (Greenwich Mean Time)
UTCDIFF	Time difference in hours (in data type FIXED(4,2)) between your local time and the UTC time value.
TRUE | FALSE	Corresponding value of a column of the data type BOOLEAN [Page 17]
TRANSACTION	Identification of the current transaction [Page 210]. This is a value of data type CHAR(10) BYTE.

Date and time format (datetimeformat)

The date and time format (datetimeformat) specifies the way in which date values [Page 17], time values [Page 17], and timestamp values [Page 17] are represented.

Syntax

```
<datetimeformat> ::= EUR | INTERNAL | ISO | JIS | USA
```

Date value

'YYYY'	Four-digit year format
'MM'	Two-digit month format (01-12)
'DD'	Two-digit day format (01-31)

Format	General Format	Example

EUR	'DD.MM.YYYY'	'23.01.1999'
INTERNAL	'YYYYMMDD'	'19990123'
ISO/JIS	'YYYY-MM-DD'	'1999-01-23'
USA	'MM/DD/YYYY'	'01/23/1999'

In all formats, with the exception of INTERNAL, leading zeros may be omitted in the identifiers for the month and day.

Time value

'HHHH'	Four-digit hour format
'HH'	Two-digit hour format
'MM'	Two-digit minute format (00-59)
'SS'	Two-digit second format (00-59)

Format	General Format	Example
EUR	'HH.MM.SS'	'14.30.08'
INTERNAL	'HHHHMMSS'	'00143008'
JIS/ISO	'HH:MM:SS'	'14:30:08'
USA	'HH:MM AM (PM)'	'2:30 PM'

In all time formats, the identifier of the hour must consist of at least one digit. In the USA time format, the minute identifier can be omitted completely. In all the other formats, with the exception of INTERNAL, the minute and second identifiers must comprise at least one digit.

Timestamp value

'YYYY'	Four-digit year format
'MM'	Two-digit month format (01-12)
'DD'	Two-digit day format (01-31)
'HH'	Two-digit hour format (0-24)
'MM'	Two-digit minute format (00-59)
'SS'	Two-digit second format (00-59)
'MMMMMM'	Six-digit microsecond format

Format	General Format	Example
EUR/JIS/USA	'YYYY-MM-DD- HH.MM.SS.MMMMMM'	'1999-01-23- 14.30.08.456234'
ISO	'YYYY-MM-DD HH:MM:SS.MMMMMM'	'1999-01-23 14:30:08.456234'
INTERNAL	'YYYYMMDDHHMMSSMMMMMM'	'19990123143008456234'

The microsecond identifier can be omitted in all timestamp formats. In all formats, with the exception of INTERNAL, the month and day identifiers must consist of at least one digit.

Explanation

The date and time format determines the format in which the date, time, and timestamp values may be represented in SQL statements and the way in which results are to be displayed.

The date and time format is determined when the database system is installed.

Users can change the date and time format for the current session by setting the relevant parameters in the database tools or by specifying the corresponding parameters when using programs.

The ISO date and time format is used by ODBC and JDBC applications and cannot be replaced with a different date and time format.

Specifying a string (string spec)

Only expressions [Page 52] that have an alphanumeric value as a result are allowed as a string specification (`string_spec`).

Specifying a Key (key_spec)

A key specification (`key_spec`) allows rows in a table to be located whose key column values match the values in the key specification. A row with the specified key values does not have to exist.

Syntax

```
<key_spec> ::= <column_name> = <value_spec>
```

column_name [Page 46], value_spec [Page 49]

Explanation

The value specification (`value_spec`) must **not** be NULL.

The column name must identify a key column in the table.

The key specification must contain all the key columns in a table. The individual key specifications (`key_spec`) must be separated by commas.

For tables defined without key columns, there is the implicitly generated column SYSKEY CHAR(8) BYTE which contains a key generated by the database system. This column can only be used in a key specification.

Expression

An expression specifies a value that is generated, if required, by applying arithmetic operators to values.

A distinction is made between the following arithmetic operators:

- Additive operators
  + Addition
  - Subtraction
- Multiplicative operators
  * Multiplication
  / Division

DIV integer division
MOD remainder after integer division

Syntax

```
<expression> ::= <term> | <expression> + <term> | <expression> -
<term>
<expression_list> ::= (<expression>,...)

<term> ::= <factor>
| <term> * <factor> | <term> / <factor>
| <term> DIV <factor>| <term> MOD <factor>
```

factor [Page 54]

Explanation

The arithmetic operators can only be applied to numeric data types.

	Result of an expression
`expression`	Value of any data type [Page 15]
`factor` supplies a NULL value	NULL value [Page 15]
`factor` supplies a special NULL value	Special NULL value [Page 15]
`expression` leads to a division by 0	Special NULL value
`expression` leads to an overflow of the internal temporary result	Special NULL value

If no parentheses are used, the operators have the following precedence:

1. The sign [Page 33] has a higher precedence than the additive and multiplicative operators.
2. The multiplicative operators have a higher precedence than the additive operators.
3. The multiplicative operators have different priorities.
4. The additive operators have different priorities.
5. Operators with the same precedence are evaluated from left to right.

Operands are fixed point numbers

Operand1 (a)	**Operand2 (b)**	**Result**
Fixed point number [Page 16] (p precision s number of decimal places	Fixed point number (p' precision s' number of decimal places)	Fixed point number (p" precision s" number of decimal places) or floating point number

The data type of the result depends on the operation as well as on the precision and number of decimal places of the operands.

Note that the data type of a column determines its name, and not the precision and number of decimal places in the current value.

Operands are fixed point numbers, operands are +, - , * or /

The result of addition, subtraction, and multiplication is generated from a temporary result which can have more than 38 valid digits. If the temporary result has no more than 38 valid digits, the final result is equal to the temporary result. Otherwise, a result is generated as a floating point number with 38 places. Decimal places are truncated if necessary.

Condition	Operator	Result
max(p-s,p'-s')	+, -	Fixed point number p''=max(p-s,p'-s')+max(s,s')+1 s''=max(s,s')
(p+p')<=38	*	Fixed point number p''=p+p' s''=s+s'
(p-s+s')<=38	/	Fixed point number p''=38 s''=38-(p-s+s') Special NULL value, if b=0

Operands are integers, operators are DIV, MOD

Condition	Operator	Result
ABS [Page 74](a)<1E38 and ABS(b)<1E38 and b<>0	DIV	TRUNC [Page 79](a/b)
b=0	DIV	Special NULL value
ABS(a)>=1E38 and b<>0 or ABS(b)>=1E38	DIV	Error message
ABS(a)<1E38 and ABS(b)<1E38 and b<>0	MOD	a-b*(a DIV b)
b=0	MOD	a
ABS(a)>=1E38 and b<>0 or ABS(b)>=1E38	MOD	Error message

An operand is a floating point number

If an operand is a floating point number [Page 16], the result of the arithmetic operation is a floating point number.

factor

Specifies how the values are determined that are to be linked in an expression [Page 52] by means of arithmetic operators.

Syntax

```
<factor> ::= [<sign>] <value_spec> | [<sign>] <column_spec>
| [<sign>] <function_spec> | [<sign>] <set_function_spec>
| <scalar_subquery> |<expression>
```

sign [Page 33], value_spec [Page 49], column_spec [Page 47], function_spec [Page 73], set_function_spec [Page 105], scalar_subquery [Page 201], expression [Page 52]

Predicate (predicate)

A predicate is specified in a WHERE condition [Page 198] in a statement which is "true", "false", or "unknown". The result is generated by applying the predicate to a specific row in a result table (see result table name [Page 41]) or to a group of rows in a table that was formed by the GROUP clause [Page 199].

Syntax

```
<predicate> ::=
  <between_predicate> | <bool_predicate> | <comparison_predicate>
| <default_predicate> | <exists_predicate> | <in_predicate>
| <join_predicate> | <like_predicate> | <null_predicate>
| <quantified_predicate> | <rowno_predicate> | <sounds_predicate>
```

between_predicate [Page 56], bool_predicate [Page 57], comparison_predicate [Page 58], default_predicate [Page 60], exists_predicate [Page 60], in_predicate [Page 61], join_predicate [Page 62], like_predicate [Page 64], null_predicate [Page 67], quantified_predicate [Page 67], rowno_predicate [Page 69], sounds_predicate [Page 70]

Explanation

- Columns in a table with the same code attribute are comparable.
- Columns with different code attributes ASCII and EBCDIC (see code tables [Page 18]) can be compared.
- Columns with the code attributes ASCII and EBCDIC can be compared with date values [Page 17], time values [Page 17], or timestamp values [Page 17].
- LONG columns [Page 16] can only be used in the NULL predicate.

Example table: customer [Page 111]

Selection without a condition:

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

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

Hollywood	Brown	Rose

Selection with restricting condition:

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

CITY	NAME	FIRSTNAME
Los Angeles	Porter	Martin
Los Angeles	Peters	Sally
Los Angeles	Randolph	Frank
Los Angeles	Peters	Joseph
Los Angeles	Brown	Susan
Los Angeles	Jackson	Anthony
Los Angeles	Adams	Thomas
Los Angeles	TOOLware	?

BETWEEN predicate (between_predicate)

The BETWEEN predicate [Page 55] (`between_predicate`) checks whether a value is within a specified range.

Syntax

```
<between_predicate> ::= <expression> [NOT] BETWEEN <expression> AND
<expression>
```

expression [Page 52]

Explanation

Let x, y, and z be the results of the first, second, and third expression. The values x,y,z must be comparable.

	Result of the specified predicate
x BETWEEN y AND z	x>=y AND x<=z
x NOT BETWEEN y AND z	NOT(x BETWEEN y AND z)
x, y, or z are NULL value [Page 15]s	x [NOT] BETWEEN y AND z is undefined

Example table: customer [Page 111]

Finding customers with a credit balance between –420 and 0:

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

TITLE	NAME	CITY	ACCOUNT
Mr	Porter	Los Angeles	0.00

Mrs	Peters	Los Angeles	0.00
Mr	Brown	Hollywood	0.00
Mr	Porter	New York	0.00
Mr	Howe	New York	-315.40
Mr	Randolph	Los Angeles	0.00
Mr	Jackson	Los Angeles	0.00
Mr	Adams	Los Angeles	-416.88
Mr	Griffith	New York	0.00

You want to list the customers who have either a credit balance or a significant debit balance:

```
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	Brown	Los Angeles	-4167.79
Mr	Adams	Los Angeles	-416.88
Comp	TOOLware	Los Angeles	3770.50
Mrs	Brown	Hollywood	440.00

Boolean predicate (bool_predicate)

Boolean values (BOOLEAN [Page 17]) are compared in a boolean predicate [Page 55].

Syntax

```
<bool_predicate> ::= <column_spec> [ IS [NOT] <TRUE | FALSE>]
```

column spec [Page 47]

Explanation

If only one column specification (`column_spec`) is specified, the syntax is identical to `<column_spec> IS TRUE`.

The `column_spec` must always denote a column with the data type BOOLEAN.

The following rules apply to the result of a boolean predicate:

Column value	IS TRUE	IS NOT TRUE	IS FALSE	IS NOT FALSE
false	false	true	true	false
undefined	undefined	undefined	undefined	undefined
true	true	false	false	true

Comparison Predicate (comparison_predicate)

A comparison predicate [Page 55] specifies a comparison between two values or lists of values.

Syntax

```
<comparison_predicate> ::= <expression> <comp_op> <expression>
| <expression> <comp_op> <subquery>
| <expression_list> <equal_or_not> (<expression_list>)
| <expression_list> <equal_or_not> <subquery>
```

expression [Page 52], expression_list [Page 52], subquery [Page 200]

The following operators are available for comparing two values:
<, >, <>, !=, =, <=, >= (comp_op [Page 59])

Value lists can only be compared with the = and <> operators (equal_or_not [Page 59]).

Explanation

The subquery (`subquery`) must supply a result table (see result table name [Page 41]) that contains the same number of columns as the number of values on the left of the operator. The result table may contain no more than one row.

The list of values specified to the right of the `equal_or_not` operator (`expression_list`) must contain the same number of values as specified in the value list in front of the `equal_or_not` operator.

The JOIN predicate [Page 62] is a special case.

Comparing two values

Let x be the result of the first expression and y the result of the second expression or of the subquery.

- The values x and y must be comparable with one another.
- Numbers are compared to one another according to their algebraic values.
- Character strings are compared character by character.
  Any blanks (code attribute [Page 17] ASCII, EBCDIC, UNICODE) or binary zeros (code attribute BYTE) at the end of one or both of the character strings are removed.
  If the character strings have the different code attributes ASCII and EBCDIC, one of the two strings is converted implicitly so that they both have the same code attribute.
  If the character strings have the different code attributes ASCII or EBCDIC and UNICODE, the character string with the code attribute ASCII or EBCDIC is implicitly converted into a character string with the code attribute UNICODE.
  Two character strings are identical if they have the same characters in all positions.
  The relationship between two character strings that are not identical is defined by the first character that differs in a comparison from left to right. This comparison is performed in accordance with the code attribute selected for this column (ASCII, EBCDIC, UNICODE, or BYTE).
- If x or y are NULL value [Page 15]s, or if the result of the subquery is empty, (x `<comp_op>` y) is not defined.

Comparing two value lists

If a value list (`expression_list`) is specified on the left of the comparison operator `equal_or_not`, x is the value list that comprises the results of the values x1, x2, ..., xn in

this list. y is the result of the subquery or the result of the second value list. A value list y consists of the results of the values y_1, y_2, ..., y_n.

- A value xm must be comparable with the associated value ym.
- x=y is true if for xm=ym for all m=1, ..., n.
- x<>y is true if at least xm<>ym for at least one m.
- (x `<equal_or_not>` y) is undefined (not known) if there is no m for which (xm `<equal_or_not>` ym) is false and if there is at least one m for which (xm `<equal_or_not>` ym) is undefined.
- If one xm or one ym is a NULL value, or if the result of the subquery is empty, (x `<equal_or_not>` y) is undefined.

Example table: customer [Page 111]

Which customers are female?

```
SELECT title, firstname, name FROM customer
WHERE title = 'Mrs'
```

TITLE	FIRSTNAME	NAME
Mrs	Jenny	Porter
Mrs	Sally	Peters
Mrs	Susan	Brown
Mrs	Rose	Brown

Comparison operators (comp_op)

Operators for comparing values (`comp_op`)

Syntax

```
<comp_op> ::= < | > | <> | != | = | <= | >=
| ¬= | ¬< | ¬> (for machines with EBCDIC code [Page 20])
| ~= | ~< | ~> (for machines with ASCII code [Page 18])
```

Comparison operators (equal_or_not)

Operators for comparing value lists (`equal_or_not`)

Syntax

```
<equal_or_not> ::= <> |=
| ¬= (for machines with EBCDIC code [Page 20])
| ~= (for machines with ASCII code [Page 18])
```

DEFAULT predicate

By specifying a DEFAULT predicate [Page 55], you can check whether a column contains the default value defined for this column.

Syntax

```
<default_predicate> ::= <column_spec> <comp_op> DEFAULT
```

column_spec [Page 47], comp_op [Page 59]

Explanation

A DEFAULT specification [Page 125] must be made for the specified column. This can be done in the following SQL statements:

- CREATE TABLE statement [Page 115]
- ALTER TABLE statement [Page 133]

If the column contains the NULL value [Page 15], `<column_spec> <comp_op> DEFAULT` is undefined.

The rules for comparing values or value lists, as defined under comparison predicate [Page 58], apply here.

EXISTS Predicate (exists_predicate)

The EXISTS predicate [Page 55] (`exists_predicate`) checks whether a result table (see result table name [Page 41]) contains at least one row.

Syntax

```
<exists_predicate> ::= EXISTS <subquery>
```

subquery [Page 200]

Explanation

The truth content of an EXISTS predicate is either true or false.

The subquery generates a result table. If this result table contains at least one row, `EXISTS <subquery>` is true.

Example tables: customer [Page 111], reservation [Page 114]

Only select customers that have one or more reservations:

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

CNO	TITLE	NAME	FIRSTNAME	ZIP	CITY	ACCOUNT
3000	Mrs	Porter	Jenny	80335	New York	100.00
3100	Comp	DATASOFT	?	50933	Dallas	4813.50
3200	Mr	Porter	Martin	10969	Los Angeles	0.00
3600	Mr	Howe	George	81737	New York	-315.40
3900	Mrs	Brown	Susan	13599	Los Angeles	-4167.79
4100	Mr	Adams	Thomas	13355	Los Angeles	-416.88

4300	Comp	TOOLware	?	13629	Los Angeles	3770.50
4400	Mrs	Brown	Rose	40233	Hollywood	440.00

IN Predicate (in_predicate)

The IN predicate [Page 55] checks whether a value or a value list is contained in a specified set of values or value lists.

Syntax

```
<in_predicate> ::=
  <expression> [NOT] IN <subquery>
| <expression> [NOT] IN <expression_list>
| <expression_list> [NOT] IN <subquery>
| <expression_list> [NOT] IN (<expression_list>,...)
```

expression [Page 52], expression_list [Page 52], subquery [Page 200]

Explanation

The subquery must supply a result table (see result table name [Page 41]) that contains the same number of columns as the number of values specified by the expression on the left-hand side of the IN operator.

Each value list specified on the right-hand side of the IN operator must contain the same number of values as specified in the value list on the left-hand side of the IN operator.

- `x [NOT] IN S`, whereby x `<expression>` and S `<subquery>` or `<expression_list>`
  The value x and the values in S must be comparable.
- `x [NOT] IN S`, whereby x `<expression_list>` with the values x1, x2, ..., xn and S `<subquery>` (set of value lists s) or `(<expression_list>,...)` (Range of values lists s) with the value lists s: s1, s2, ..., sn
  A value xm must be comparable with all values sm.
  x=s is true if xm=sm, m=1,...,n
  x=s is false if there is at least one m for which xm=sm is false
  x=s is undefined if there is no m for which xm=sm is false and there is at least one m for which xm=sm is undefined.

The entry '------' in the list below means that no statement can be made if only the result of the comparison with one s is known.

	Result of the function x IN S
x=s is true for at least one s	true
x=s is true for all s	true
S contains NULL values and x=s is true for the remaining s	true
S is empty	false
x=s is false for at least one s	------
x=s is false for all s	false
S contains NULL values and x=s is false for the remaining s	undefined
x=s is not true for any s and is undefined for at least one value s	undefined

`x NOT IN S` has the same result as `NOT(x IN S)`

Example table: customer [Page 111]

Choosing all customers who are natural persons (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	Porter	Los Angeles
Mrs	Sally	Peters	Los Angeles
Mr	Peter	Brown	Hollywood
Mr	Michael	Porter	New York
Mr	George	Howe	New York
Mr	Frank	Randolph	Los Angeles
Mr	Joseph	Peters	Los Angeles
Mrs	Susan	Brown	Los Angeles
Mr	Anthony	Jackson	Los Angeles
Mr	Thomas	Adams	Los Angeles
Mr	Mark	Griffith	New York
Mrs	Rose	Brown	Hollywood

JOIN Predicate (join_predicate)

A JOIN predicate [Page 55] specifies a JOIN. A JOIN predicate can be specified with or without one or with two OUTER JOIN indicators.

Syntax

```
<join_predicate> ::=
<expression> [<outer_join_indicator>] <comp_op> <expression>
[<outer_join_indicator>]

<outer_join_indicator> ::= (+)
```

expression [Page 52], comp_op [Page 59]

Explanation

Each `expression` must contain a column specification [Page 47]. A column specification must exist for the first and second expression so that both specifications refer to different table names or reference names.

Let x be the value of the first expression and y the value of the second expression. The values x and y must be comparable with one another.

The rules outlined under comparison predicate [Page 58] apply here.

If at least one OUTER JOIN indicator is specified in a JOIN predicate of a search condition [Page 70], the corresponding table expression must be based on exactly two tables, or the following has to apply:

- OUTER JOIN indicators are only specified for one of the tables in the FROM clause [Page 196].
- All of the JOIN predicates in this table to just one other table contain the OUTER JOIN indicator.
- All other JOIN predicates contain no OUTER JOIN indicators.

If a JOIN requires more than two tables for the QUERY specification [Page 192] and if one of the rules above cannot be observed, a QUERY expression [Page 189] can also be used in the FROM clause.

Only those rows from the table that have a counterpart of the comparison operator in the JOIN predicate specified in the table are transferred to the result table.

The OUTER JOIN indicator must be specified on the side of the comparison operator where the other table is specified if each row in a table is to appear at least once in the result table. If it is not possible to find at least one counterpart for a table row in the other table, this row is used to build a row for the result table. The NULL value is then used for the output columns which are formed from the columns in the other table.
Since the OUTER JOIN indicator can be specified on both sides of the comparison operator if the table expression [Page 195] is based on just two tables, it can be ensured that each line in both tables appears at least once in the result table.

The JOIN predicate is a special case of the comparison predicate [Page 58]. The number of JOIN predicates in a search condition is limited to 128.

JOIN predicate

Example tables: customer [Page 111], reservation [Page 114]

Is there a reservation for the customer 'Porter'? If so, for what date?

```
SELECT reservation.rno, customer.name,customer.firstname,
reservation.arrival, departure

FROM customer, reservation

WHERE customer.name = 'Porter' AND customer.cno =
reservation.cno
```

RNO	NAME	FIRSTNAME	ARRIVAL	DEPARTURE
100	Porter	Jenny	13/11/2001	15/11/2001
110	Porter	Jenny	24/12/2001	06/01/2002
120	Porter	Martin	14/11/2001	18/11/2002

Specifying an OUTER JOIN indicator

Example tables: hotel [Page 112], reservation [Page 114]

List all the hotels in Los Angeles for which a reservation exists and those for which a reservation does not exist. Missing reservation numbers are assigned a NULL value.

```
SELECT hotel.hno, hotel.name, reservation.rno

FROM hotel, reservation

WHERE hotel.city = 'Los Angeles' AND hotel.hno =
reservation.hno (+)
```

HNO	NAME	RNO
80	Midtown	100
50	Lake Michigan	120

80	Midtown	140
120	Sunshine	180
40	Eight Avenue	?

LIKE Predicate (like_predicate)

A LIKE predicate [Page 55] is used to search for character strings [Page 16] that have a particular pattern. This pattern can be a certain character string or any sequence of characters (whose length may or may not be known).

Syntax

```
<like_predicate> ::= <expression> [NOT] LIKE <like_expression>
[ESCAPE <expression>]

<like_expression> ::= <expression> | '<pattern_element>...'
```

expression [Page 52], pattern_element [Page 65]

Explanation

The expression in the `like_expression` must supply an alphanumeric value or a date or time value.

`x NOT LIKE y` has the same result as `NOT(x LIKE y)`.

	Result of x LIKE y
x or y are NULL value [Page 15]s	x LIKE y is undefined
x and y are non-NULL values	x LIKE y is either true or false
x can be split into substrings with the result that: A substring of x is a sequence of 0,1, or more contiguous characters, and each character of x belongs to exactly one substring. The number of substrings of x and y is identical. If the nth pattern element of y is a sequence of characters and the nth substring of x is a sequence of 0 or more characters.	x LIKE y is true

ESCAPE

An escape character (`ESCAPE <expression>`) must be used if a search is to be performed for an <underscore [Page 37]>, '%', or the hexadecimal value X'1E' or X'1F' in the LIKE predicate.

If `ESCAPE <expression>` is specified, the corresponding expression (`expression`) must supply an alphanumeric value that consists of just one character. If this escape character is contained in the LIKE expression (`like_expression`), the following character must be one of the special characters <underscore>, %, X'1E', or X'1F'. This special character is then viewed as standing for itself.

Search for any character string with a minimum length of 1: `LIKE '%_'`
Search for a character string in which a fixed number of characters is known: `LIKE'_c_'`

Search for a character string with any number of characters, whereby the character string must contain an <underscore>: `LIKE '%:_%'ESCAPE':'`

Example table: customer [Page 111]

Customers whose name ends with 'FT':

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

NAME	CITY
DATASOFT	Dallas

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

```
SELECT name, firstname, city FROM customer
WHERE name LIKE 'P_ _ _ _ _'
```

NAME	FIRSTNAME	CITY
Porter	Jenny	New York
Porter	Martin	Los Angeles
Peters	Sally	Los Angeles
Porter	Michael	New York
Peters	Joseph	Los Angeles

Customers with a 'p' in their name from the second position:

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

NAME	CITY
Randolph	Los Angeles

Pattern element

Element for specifying a comparison pattern (for a LIKE predicate [Page 64]). A comparison can be carried out with a string of characters or a set of characters.

Syntax

```
<pattern_element> ::= <match_string> | <match_set>
```

match_string [Page 65], match_set [Page 66]

match_string

If a `match_string` is specified, this position in the search pattern can be replaced by any number of characters.

Syntax

```
<match_string> ::= % | X'1F'
```

Explanation

A LIKE predicate [Page 64] is used to search for character strings [Page 16] that have a certain pattern. `Match_string` can be used to specify the pattern (pattern element [Page 65]).

Example table: customer [Page 111]

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

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

FIRSTNAME	NAME
Martin	Porter
Michael	Porter
Mark	Griffith

match_set

If a `match_set` is specified, this position in the search pattern can be replaced by the exact number of characters specified in the `match_set`.

Syntax

```
<match_set> ::= <underscore> | X'1E' | <match_char>
```

`<match_char>` ::= Every character [Page 30] other than %, X'1F', underscore, X'1E'

underscore [Page 37]

Explanation

A LIKE predicate [Page 64] is used to search for character strings [Page 16] that have a certain pattern. `Match_set` can be used to specify the pattern (pattern element [Page 65]).

- `<underscore> | X'1E'` : this position in the pattern can be replaced by any one character
- `match_char` : this position in the pattern can be replaced by the specified character itself.

Example table: customer [Page 111]

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

```
SELECT name, firstname, city FROM customer
WHERE name LIKE 'P_ _ _ _ _'
```

NAME	FIRSTNAME	CITY
Porter	Jenny	New York

Porter	Martin	Los Angeles
Peters	Sally	Los Angeles
Porter	Michael	New York
Peters	Joseph	Los Angeles

NULL predicate

By specifying a NULL predicate [Page 55], you can test whether the value is a NULL value [Page 15].

Syntax

```
<null_predicate> ::= <expression> IS [NOT] NULL
```

expression [Page 52]

Explanation

The truth content of a NULL predicate is either true or false.

	Result of the function x IS NULL
x is NULL value	true
x is a special NULL value [Page 15]	false

`x IS NOT NULL` has the same result as `NOT(x IS NULL)`.

Quantified Predicate (quantified_predicate)

By specifying a quantity predicate [Page 55], you can compare a value or list of values to a set of values or value lists.

Syntax

```
<quantified_predicate> ::=
  <expression> <comp_op> <quantifier> <expression_list>
| <expression> <comp_op> <quantifier> <subquery>
| <expression_list> <equal_or_not> <quantifier>
(<expression_list>,...)
| <expression_list> <equal_or_not> <quantifier> <subquery>
```

subquery [Page 200], expression_list [Page 52]

The following operators are available for comparing values:
<, >, <>, !=, =, <=, >= (comp op [Page 59])

Value lists can only be compared with the = and <> operators (equal_or_not [Page 59]).

The quantified predicate can be qualified with ALL, SOME, or ANY (quantifier [Page 69]).

Explanation

The subquery must supply a result table (see result table name [Page 41]) that contains the same number of columns as the number of values specified by the expression or expression list on the left-hand side of the operator.

Each list of values specified to the right of the `equal_or_not` operator (`expression_list`) must contain the same number of values as specified in the value list in front of the `equal_or_not` operator.

- Let x be the result of the first expression and S the result of the subquery or sequence of values. S is a set of values s. The value x and the values in S must be comparable with each other.
- If a value list (`expression_list`) is specified on the left of the comparison operator `equal_or_not`, then let x be the value list comprising the results of the values x_1, x_2, ..., x_n of this value list. Let S be the result of the subquery consisting of a set of value lists s or a sequence of value lists s. A value list s consists of the results of the values s_1, s_2, ..., s_n. A value xm must be comparable with all values sm.
  x=s is true if xm=sm, m=1,...,n
  x<>s is true if there is at least one m for which xm<>sm
  x `<equal or not>` s is undefined if there is no m for which xm `<equal or not>` sm is false and if there is at least one m for which xm `<equal_or_not>` sm is undefined. If one xm or one ym is a NULL value, or if the result of the subquery is empty, x `<equal_or_not>` y is undefined.

The entry '------' in the list below means that no statement can be made if only the result of the comparison with one s is known.

x <compare> <quantifier> S, whereby compare ::= comp_op | equal_or_not

	quantifier ::= ALL	**quantifier ::= ANY | SOME**
S is empty	true	false
`x <compare> S` is true for at least one s from S	------	true
`x <compare> S` is true for all s from S	true	true
`x <compare> S` is not false for any value from S and is undefined for at least one value s	undefined	
S contains NULL values and `x <compare> S` is true for all other s	undefined	true
`x <compare> S` is false for at least one value s from S	false	------
`x <compare> S` is false for all s from S	false	false
`x <compare> S` is not true for any value s from S and is undefined for at least one value s		undefined
S contains NULL values and `x <compare> S` is false for all other s	false	undefined

Example table: hotel [Page 112]

List of hotels that have the same name as other cities in the base table.

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

The subquery `SELECT city FROM hotel` determines the list of city names that are compared with the hotel names.

NAME	CITY
Los Angeles	Cincinatti
Long Beach	Long Beach
Dallas	Dallas

Quantifier

The quantified predicate [Page 67] can be qualified with ALL, SOME, or ANY.

Syntax

```
<quantifier> ::= ALL | SOME | ANY
```

ROWNO Predicate (rowno_predicate)

The ROWNO predicate [Page 55] restricts the number of lines in a result table (see result table name [Page 41]).

Syntax

```
<rowno_predicate> ::= ROWNO < <unsigned_integer>
| ROWNO < <parameter_spec>
| ROWNO <= <unsigned_integer>
| ROWNO <= <parameter_spec>
```

unsigned_integer [Page 34], parameter_spec [Page 48]

Explanation

A ROWNO predicate may only be used in a WHERE clause [Page 198] that belongs to a QUERY statement. The ROWNO predicate can be used like any other predicate [Page 55] in the WHERE clause if the following restrictions are observed:

- The ROWNO predicate must be linked to the other predicates by a logic AND
- The ROWNO predicate must not be negated
- The ROWNO predicate may not be used more than once in the WHERE clause

You can specify the maximum number of lines in the result table using an unsigned integer or a parameter specification. If more lines are found, they are simply ignored and do not lead to an error message. Specifying a ROWNO predicate of the type ROWNO <= 0 results in an empty results table.

If a ROWNO predicate and an ORDER clause [Page 202] are specified, only the first n result lines are searched and sorted. The result usually differs from that which would have been obtained if a ROWNO predicate had not been used and if the first n result rows had been considered.

If a ROWNO predicate and a set function [Page 105] are specified, the set function is only applied to the number of lines restricted by the ROWNO predicate.

SOUNDS predicate

A SOUNDS predicate [Page 55] is used to perform a phonetic comparison.

Syntax

```
<sound_predicate> ::= <expression> [NOT] SOUNDS [LIKE] <expression>
```

expression [Page 52]

Explanation

Specifying LIKE in the SOUNDS predicate has no effect.

The values in the expressions must be alphanumeric (code attribute ASCII, EBCDIC, see code tables [Page 18]).

A phonetic comparison between values is carried out according to the SOUNDEX algorithm. First, all vowels and some consonants are eliminated, then all consonants which are similar in sound are mapped to each other.

x [NOT] SOUNDS [LIKE] y

	Result of the predicate
x or y is the NULL value [Page 15]	`x SOUNDS y` is undefined
x and y are non-NULL values	`x SOUNDS y` is true or false
x and y are phonetically identical	`x SOUNDS y` is true

`x NOT SOUNDS y` has the same result as `NOT (x SOUNDS y)`.

See also:

SOUNDEX(x) [Page 88] string function

Search Condition (search_condition)

A `search_condition` links statements that can be true, false, or undefined. Rows in a table may be found that fulfill several conditions that are linked with AND or OR.

Syntax

```
<search_condition> ::= <boolean_term> | <search_condition> OR
<boolean_term>
```

```
<boolean_term> ::= <boolean_factor> | <boolean_term> AND
<boolean_factor>
```

boolean factor [Page 72]: determine the boolean values (BOOLEAN [Page 17]) to be linked or their negation (`NOT`).

Explanation

Predicates in a WHERE clause [Page 198] are applied to the specified row or a group of rows in a table formed with the GROUP clause [Page 199]. The results are linked using the specified Boolean operators (AND, OR, NOT).

If no parentheses are used, the precedence of the operators is as follows: NOT has a higher precedence than AND and OR, AND has a higher precedence than OR. Operators with the same precedence are evaluated from left to right.

NOT

x	NOT(x)
true	false
false	true
undefined	undefined

x AND y

x y	false	undefined	true
false	false	false	false
undefined	false	undefined	undefined
true	false	undefined	true

x OR y

x y	false	undefined	true
false	false	undefined	true
undefined	undefined	undefined	true
true	true	true	true

Example table: customer [Page 111]

Customers who live in New York or have a credit balance:

```
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
Michael	Porter	New York	0.00
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

Customers who live in Hollywood and have a credit balance:

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

FIRSTNAME	NAME	CITY	ACCOUNT
Rose	Brown	Hollywood	440.00

Boolean factor

Specifies how the Boolean values are determined that are to be linked in a search condition [Page 70] by AND or OR.

Syntax

```
<boolean_factor> ::= [NOT] <predicate> | [NOT] (<search_condition>)
```

predicate [Page 55], search_condition [Page 70]

Functions: Overview

List of all function names

ABS [Page 74]	ACOS [Page 80]	ADDDATE [Page 93]
ADDTIME [Page 95]	ALPHA [Page 81]	ASCII [Page 82]
ASIN [Page 80]	ATAN [Page 80]	ATAN2 [Page 80]
AVG [Page 108]		
CASE [Page 101]	CEIL [Page 74]	CHAR [Page 103]
CHR [Page 104]	COS [Page 80]	COSH [Page 80]
COT [Page 80]	COUNT [Page 108]	
DATE [Page 97]	DATEDIFF [Page 93]	DAY [Page 99]
DAYNAME [Page 94]	DAYOFMONTH [Page 94]	DAYOFWEEK [Page 94]
DAYOFYEAR [Page 94]	DECODE [Page 100]	DEGREES [Page 80]
EBCDIC [Page 82]	EXP [Page 74]	EXPAND [Page 82]
FIXED [Page 75]	FLOAT [Page 75]	FLOOR [Page 75]
GET_OBJECTNAME [Page 82]	GET_OWNER [Page 83]	GREATEST [Page 100]
HEX [Page 104]	HEXTORAW [Page 104]	HOUR [Page 98]
INDEX [Page 75]	INITCAP [Page 84]	
LEAST [Page 100]	LENGTH [Page 76]	LFILL [Page 84]
LN [Page 77]	LOG [Page 77]	LOWER [Page 91]
LPAD [Page 85]	LTRIM [Page 86]	
MAKEDATE [Page 94]	MAKETIME [Page 96]	MAX [Page 109]
MAPCHAR [Page 86]	MICROSECOND [Page 98]	MIN [Page 109]
MINUTE [Page 98]	MONTH [Page 99]	MONTHNAME [Page 94]
NOROUND [Page 78]	NUM [Page 105]	
PI [Page 78]	POWER [Page 78]	
RADIANS [Page 80]	REPLACE [Page 86]	RFILL [Page 87]
ROUND [Page 78]	RPAD [Page 88]	RTRIM [Page 88]
SECOND [Page 98]	SIGN [Page 79]	SIN [Page 80]

SINH [Page 80]	SOUNDEX [Page 88]	
SQRT [Page 79]	SUBDATE [Page 93]	SUBSTR [Page 89]
SUBTIME [Page 95]	STDDEV [Page 109]	SUM [Page 109]
TAN [Page 80]	TANH [Page 80]	TIME [Page 98]
TIMEDIFF [Page 96]	TIMESTAMP [Page 98]	TRANSLATE [Page 90]
TRIM [Page 90]	TRUNC [Page 79]	
UPPER [Page 91]		
VALUE [Page 101]	VARIANCE [Page 109]	
WEEKOFYEAR [Page 94]		
YEAR [Page 99]		
|| [Page 91]	& [Page 91]	

Function (function_spec)

There is a series of functions that can be applied to a value (row) as an argument (`function_spec`) and supply a result.

Syntax

```
<function_spec> ::= <arithmetic_function> | <trigonometric_function>
| <string_function> | <date_function> | <time_function>
| <extraction_function> | <special_function>
| <conversion_function>
```

arithmetic_function [Page 73], trigonometric_function [Page 80], string_function [Page 81], date_function [Page 92], time_function [Page 95], extraction_function [Page 97], special_function [Page 99], conversion_function [Page 103]

Explanation

The arguments and results of the functions are numeric, alphanumeric or Boolean values that are subject to certain restrictions. LONG columns [Page 16] are not allowed as arguments.

Arithmetic function

An arithmetic function is a function [Page 73] that supplies a numeric value as a result.

Syntax

```
<arithmetic_function> ::=
  TRUNC    ( <expression>[, <expression>] )
| ROUND    ( <expression>[, <expression>] )
| NOROUND  ( <expression> )
| FIXED    ( <expression>[, <unsigned_integer> [, <unsigned_integer] ]
)
| FLOAT    ( <expression>[, <unsigned_integer> ] )
| CEIL     ( <expression> )
| FLOOR    ( <expression> )
| SIGN     ( <expression> )
| ABS      ( <expression> )
| POWER    ( <expression>, <expression> )
| EXP      ( <expression> )
```

```
| SQRT     ( <expression> )
| LN       ( <expression> )
| LOG      ( <expression>, <expression> )
| PI
| LENGTH   ( <expression> )
| INDEX    ( <string_spec>, <string_spec> [,<expression>[,
<expression>] ] )
```

expression [Page 52], unsigned_integer [Page 34], string_spec [Page 52]

TRUNC(a,n) [Page 79], ROUND(a,n) [Page 78], NOROUND(a) [Page 78], FIXED(a,p,s) [Page 75], FLOAT(a,s) [Page 75], CEIL(a) [Page 74], FLOOR(a) [Page 75], SIGN(a) [Page 79], ABS(a) [Page 74], POWER(a,n) [Page 78], EXP(a) [Page 74], SQRT(a) [Page 79], LN(a) [Page 77], LOG(a,b) [Page 77], PI [Page 78], LENGTH(a) [Page 76], INDEX(a,b,p,s) [Page 75]

ABS(a)

ABS(a) is an arithmetic_function [Page 73] that determines the unsigned value (absolute value) of the number a.

	Result of the ABS(a) function
a is NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

CEIL(a)

CEIL(a) is an arithmetic function [Page 73] that calculates the smallest integer value that is greater than or equal to the number a.

The result is a fixed point number with 0 decimal places.

An error message is output if it is not possible to represent the result of CEIL(a) as a fixed point number.

	Result of CEIL(a) function
a is the NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

EXP(a)

EXP(a) is an arithmetic function [Page 73] that calculates the power from base e (2.71828183) and the exponent a ("e to the power of a").

	Result of EXP(a) function
a is NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

FIXED(a,p,s)

FIXED(a,p,s) is an arithmetic function [Page 73] that is used to output the number a in a format of the data type FIXED [Page 121](p,s).

Digits after the decimal point are rounded to s decimal places, if necessary.

	Result of the FIXED(a,p,s) function
s not specified	Result as for s=0
p not specified	Result as for p=38
ABS(a)>10exp(p-s)	Special NULL value [Page 15]
a is the NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value

FLOAT(a,s)

FLOAT(a,s) is an arithmetic function [Page 73] that outputs the figure a in a format of the data type FLOAT [Page 122](s). It is rounded to s places if necessary.

	Result of the FLOAT(a,s) function
a is NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

FLOOR(a)

FLOOR(a) is an arithmetic function [Page 73] that calculates the largest integer value that is less than or equal to the number a.

The result is a fixed point number with 0 decimal places.

An error message is output if it is not possible to represent the result of FLOOR(a) as a fixed point number.

	Result of the FLOOR(a) function
a is the NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

INDEX(a,b,p,s)

INDEX(a,b,p,s) is an arithmetic function [Page 73] that determines the position of the substring specified in b within the character string a.

The parameter p is optional. If p is specified (p>=1), it defines the start position for the search for the substring b. If p is not specified, the search is started at position 1.

The parameter s is optional. If s is specified, it determines the number of searches for the substring b. If s is not specified, the search is carried out for the first occurrence.

	Result of the INDEX(a,b,p,s) function

a, b character strings and b not less than s times substring of a	0
a character string and b empty character string	p
a,b,p or s is NULL value	NULL value [Page 15]
p or s is special NULL value [Page 15]	Error Message

Example table: customer [Page 111]

The position of the character string ‘er’ is to be determined in all customer surnames.

```
SELECT name, INDEX(name,'er') position_er FROM customer
```

NAME	POSITION_ER
Porter	5
DATASOFT	0
Porter	5
Peters	4
Brown	0
Porter	5
Howe	0
Randolph	0
Peters	4
Brown	0
Jackson	0
Adams	0
Griffith	0
TOOLware	0
Brown	0

LENGTH(a)

LENGTH(a) is an arithmetic function [Page 73] that specifies the number of characters or bytes that are required to represent the value a internally. The function can be used for all data types except LONG [Page 121].

	Result of the LENGTH(a) function
a is a value of data type CHAR [Page 120]|VARCHAR [Page 120] <ASCII|EBCDIC|UNICODE> with n characters	Number of characters n
a is a value of another data type (other than LONG) of length n	Length n in bytes

a is a NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

The length of the number of characters or the length in bytes is determined without any consideration of trailing blanks (code attribute ASCII, EBCDIC, UNICODE) or binary zeros (code attribute BYTE).

Example table: customer [Page 111]

The `customer` table is sorted according to the length of the surnames, with names with the same length sorted in alphabetical order.

```
SELECT name, LENGTH(name) mylength
FROM customer ORDER BY mylength, name
```

NAME	MYLENGTH
Howe	4
Adams	5
Brown	5
Brown	5
Brown	5
Peters	6
Peters	6
Porter	6
Porter	6
Porter	6
Jackson	7
DATASOFT	8
Griffith	8
Randolph	8
TOOLware	8

LN(a)

LN(a) is an arithmetic function [Page 73] that calculates the natural logarithm of the number a.

	Result of the LN(a) function
a is NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

LOG(a,b)

LOG(a,b) is an arithmetic function [Page 73] that calculates the logarithm of the number b to base a.

	Result of the LOG(a,b) function
a or b is the NULL value	NULL value [Page 15]
b is special NULL value	Special NULL value [Page 15]

NOROUND(a)

NOROUND(a) is an arithmetic function [Page 73] that prevents the result of an UPDATE or INSERT statement from being rounded so that it matches the data type of the target column.

If the non-rounded number does not correspond to the data type of the target column, an error message is output.

	Result of the NOROUND(a) function
a is the NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

PI

PI is an arithmetic function [Page 73] that outputs the value π.

POWER(a,n)

POWER(a,n) is an arithmetic function [Page 73] that calculates the nth power of the number a.

An error message is output if n is not an integer.

	Result of the POWER(a,n) function
a or n is the NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

ROUND(a,n)

ROUND(a,n) is an arithmetic function [Page 73] with the following result (for the values a and n):

ROUND(a) – round decimal places up and down
ROUND(a,n) – round up and down to the nth place on the right of the decimal point
ROUND(a,-n) – round up and down to the nth place on the left of the decimal point

	Result of the ROUND(a,n) function
a>=0	TRUNC [Page 79](a+0.5*10E-n,n)
a<0	TRUNC(a-0.5*10E-n,n)
n not specified	Result as for n=0
n is not an integer	The integer component of n is used and the result is as with a>=0 or a<0
a is floating point number	Floating point number
a is fixed point number	Fixed point number

a is the NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

SIGN(a)

SIGN(a) is an arithmetic function [Page 73] that indicates the sign of the number a.

	Result of the SIGN(a) function
a<0	-1
a=0	0
a>0	1
a is the NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

SQRT(a)

SQRT(a) is an arithmetic function [Page 73] that calculates the square root of the number a.

	Result of the SQRT(a) function
a>0	Square root of a
a=0	0
a<0 or a is NULL value	NULL value [Page 15]
a is special NULL value	Special NULL value [Page 15]

TRUNC(a,n)

TRUNC(a,n) is an arithmetic function [Page 73] with the following result (for the values a and n):

TRUNC(a) – truncate the decimal places of a
TRUNC(a,n) – truncate the number a after n decimal places
TRUNC(a,-n) – set n places in the number a before the decimal point to 0

	Result of the TRUNC(a,n) function
n>0	Number a that is truncated n places after the decimal point
n=0	Integer component of a
n<0	Number a that is truncated s places in front of the decimal point
n not specified	As with n=0
n is not an integer	The integer component of n is used and the result is as with n>0, n=0, or n<0
a is floating point number	Floating point number [Page 16]
a is fixed point number	Fixed point number [Page 16]
a is the NULL value	NULL value [Page 15]

a is special NULL value	Special NULL value [Page 15]

Trigonometric function

A trigonometric function is a function [Page 73] that supplies a numeric value as a result.

Syntax

```
<trigonometric_function> ::=
  COS      ( <expression> )
| SIN      ( <expression> )
| TAN      ( <expression> )
| COT      ( <expression> )
| COSH     ( <expression> )
| SINH     ( <expression> )
| TANH     ( <expression> )
| ACOS     ( <expression> )
| ASIN     ( <expression> )
| ATAN     ( <expression> )
| ATAN2    ( <expression>, <expression> )
| RADIANS  ( <expression> )
| DEGREES  ( <expression> )
```

expression [Page 52]

All of the values (`expression`) in every trigonometric function identify an angle in radians. The only exception to this is the RADIANS function.

	Result of the trigonometric function
`<expression>` is NULL value	NULL value [Page 15]
`<expression>` is special NULL value	Special NULL value [Page 15]
`COS(a)`	Cosine of number a
`SIN(a)`	Sine of number a
`TAN(a)`	Tangent of number a
`COT(a)`	Cotangent of number a
`COSH(a)`	Hyperbolic cosine of number a
`SINH(a)`	Hyperbolic sine of number a
`TANH(a)`	Hyperbolic tangent of number a
`ACOS(a)`	Arc cosine of number a
`ASIN(a)`	Arc sine of number a
`ATAN(a)`	Arc tangent of number a
`ATAN2(a,b)`	Arc tangent of the value a/b, where -π<=a<=+π and -π<=b<=+π
`ASIN(a)`	Arc sine of number a
`RADIANS(a)`	Angle in radians of the number a
`DEGREES(a)`	Value in degrees of the number a

String Function (string_function)

A string function is a function [Page 73] that supplies an alphanumeric value as a result.

Syntax

```
<string_function> ::=
  <string_spec> || <string_spec>
| <string_spec> & <string_spec>
| SUBSTR    (<string_spec>,<expression>[,<expression>])
| LFILL     (<string_spec>,<string_literal>[,<unsigned_integer>])
| RFILL     (<string_spec>,<string_literal>[,<unsigned_integer>])
| LPAD      (<string_spec>,<expression>,<string_literal>[,<unsigned_i
nteger>])
| RPAD      (<string_spec>,<expression>,<string_literal>[,<unsigned_i
nteger>])
| TRIM      (<string_spec>[,<string_spec>])
| LTRIM     (<string_spec>[,<string_spec>])
| RTRIM     (<string_spec>[,<string_spec>])
| EXPAND    (<string_spec>,<unsigned_integer>)
| UPPER     (<string_spec>)
| LOWER     (<string_spec>)
| INITCAP   (<string_spec>)
| REPLACE   (<string_spec>,<string_spec>[,<string_spec>])
| TRANSLATE (<string_spec>,<string_spec>,<string_spec>)
| MAPCHAR   (<string_spec>[,<unsigned_integer>][,<mapchar_set_name>])
| ALPHA     (<string_spec>[,<unsigned_integer>])
| ASCII     (<string_spec>)
| EBCDIC    (<string_spec>)
| SOUNDEX   (<string_spec>)
| GET_OBJECTNAME (<string_literal>)
| GET_OWNER      (<string_literal>)
```

string_spec [Page 52], expression [Page 52], string_literal [Page 32], unsigned_integer [Page 34], mapchar_set_name [Page 42]

Concatenation (x||y and x&y) [Page 91], SUBSTR(x,a,b) [Page 89], LFILL(x,a,n) [Page 84], RFILL(x,a,n) [Page 87], LPAD(x,a,y,n) [Page 85], RPAD(x,a,y,n) [Page 88], TRIM(x,y) [Page 90], LTRIM(x,y) [Page 86], RTRIM(x,y) [Page 88], EXPAND(x,n) [Page 82], UPPER(x)/LOWER(x) [Page 91], INITCAP(x) [Page 84], REPLACE(x,y,z) [Page 86], TRANSLATE(x,y,z) [Page 90], MAPCHAR(x,n,i) [Page 86], ALPHA(x,n) [Page 81], ASCII(x)/EBCDIC(x) [Page 82], SOUNDEX(x) [Page 88], GET_OBJECTNAME(x,o) [Page 82], GET_OWNER(x,o) [Page 83]

ALPHA(x,n)

ALPHA(x,n) is a string function [Page 81] that enables a character x in ASCII or EBCDIC code (code tables [Page 18]) to be converted to a different one or two-character representation in the DEFAULTMAP (mapchar_set_name [Page 42]). ALPHA(x,n) is used to define the sort sequence.

The function ALPHA(x,n) uses the MAPCHAR(x,n,i) [Page 86] function internally (where i is the DEFAULTMAP) and also performs a conversion to uppercase letters (UPPER(x) [Page 91]).

The parameter n is optional and specifies the maximum length of the result.

	Result of the ALPHA(x,n) function

ALPHA(x,n)	UPPER(MAPCHAR(x,n,DEFAULTMAP))

The function ALPHA enables an appropriate sort, e.g. if "ü" is to be treated for as "UE" sorting purposes. The MAPCHAR SET with the name DEFAULTMAP is used.

```
SELECT...,ALPHA(<column name>) sort,...FROM...ORDER BY sort
```

ASCII/EBCDIC(x)

ASCII(x) or EBCDIC(x) is a string function [Page 81] that converts a character string [Page 16] x in ASCII code [Page 18] to EBCDIC code [Page 20] and vice versa.

	Result of the ASCII(x) or EBCDIC(x) function
Code attribute for x is ASCII or EBCDIC	ASCII(x) is a character string in ASCII notation
Code attribute for x is ASCII or EBCDIC	EBCDIC(x) is a character string in EBCDIC notation
x is NULL value	NULL value [Page 15]

The ASCII and EBCDIC functions are useful when a specific code is to be used for sorting or comparison purposes.

Output of a sorted result table in EBCDIC order:

```
SELECT EBCDIC(name) FROM...WHERE...ORDER BY 1
```

EXPAND(x,n)

EXPAND(x,n) is a string function [Page 81] that inserts as many blanks (code attribute ASCII, EBCDIC (code tables [Page 18])) or binary zeros (code attribute BYTE) to the end of a character string [Page 16] x as are needed to give the string the length specified by n.

	Result of the EXPAND(x,n) function
x is NULL value	NULL value [Page 15]

See also:

LFILL(x,a,n) [Page 84]

RFILL(x,a,n) [Page 87]

GET_OBJECTNAME(x,o)

The string function [Page 81] GET_OBJECTNAME(x,o) returns the name of the database object.

x: Identifier of the object
The argument x is a string literal [Page 32] in the format `[<owner>.]<identifier>` or a

character parameter of this type.
<owner> and <identifier> [Page 36] can be special identifiers [Page 37].

o: Object type
You can use one of the following key words:
DBPROC[EDURE]: database procedure [Page 28]
DOMAIN: domain [Page 24]
SEQUENCE: sequence [Page 45]
SYNONYM: synonym [Page 24]
VIEW: view table [Page 23]
TABLE: table [Page 23]

	Result of the Function GET_OBJECTNAME(x,o)
x identifies an object that is not of object type o	NULL value [Page 15]
Object x does not exist	NULL value
Object x exists, however the current user does not have any privileges for this object	NULL value
Object x exists for the owner specified under <owner> and the current user has a privilege for this object	<identifier>
Object x exists, and the current user has a privilege for this object	First object name found in the search hierarchy

For some object types, there is no search hierarchy. In this case, the function only searches for the object in the schema of the current user.

GET_OWNER(x,o)

The string function [Page 81] GET_OWNER(x,o) returns the name of the owner [Page 41] of the specified database object.

x: Identifier of the object
The argument x is a string literal [Page 32] in the format [<owner>.]<identifier> or a character parameter of this type.
<owner> and <identifier> [Page 36] can be special identifiers [Page 37].

o: Object type
You can use one of the following key words:
DBPROC[EDURE]: database procedure [Page 28]
DOMAIN: domain [Page 24]
SEQUENCE: sequence [Page 45]
SYNONYM: synonym [Page 24]
VIEW: view table [Page 23]
TABLE: table [Page 23]

	Result of the Function GET_OWNER(x,o)
x identifies an object that is not of object type o	NULL value [Page 15]
Object x does not exist	NULL value
Object x exists, however the current user does not have any privileges for this object	NULL value
Object x exists for the owner specified under <owner> and the current user has a privilege for	<owner>

this object	
Object x exists, and the current user has a privilege for this object	First owner of this object found in the search hierarchy

For some object types, there is no search hierarchy. In this case, the function only searches for the object in the schema of the current user.

INITCAP(x)

INITCAP(x) is a string function [Page 81] that changes a character string [Page 16] in such a way that the first character of a word is an uppercase letter and the rest of the word consists of lowercase characters. Words are separated by one or more characters which are neither letters [Page 30] nor digits [Page 30].

	Result of the Function INITCAP(x)
x is NULL value	NULL value [Page 15]

Example table: customer [Page 111]

Standardizing the notation for names

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

NAME	NEW_NAME
DATASOFT	Datasoft
TOOLware	TOOLware

LFILL(x,a,n)

LFILL(x,a,n) is a string function [Page 81] that inserts the character string a at the start of the character string [Page 16] x as often as required until the character string has reached the length n. If the character string a cannot be completely inserted without exceeding the specified total length, only the part that is needed to reach the total length is inserted.

	Result of the LFILL(x,a,n) function
LFILL(x,a) x must identify a CHAR or VARCHAR column.	The column x is filled with the characters of a until its maximum length is reached.
x is NULL value	NULL value [Page 15]
a or n is the NULL value	Error Message

Example table: customer [Page 111]

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

FIRSTNAME	NAME	CITY
Martin	Porter	Los Angeles

Sally	Peters	Los Angeles
Frank	Randolph	Los Angeles
Joseph	Peters	Los Angeles
Susan	Brown	Los Angeles
Anthony	Jackson	Los Angeles
Thomas	Adams	Los Angeles

RFILL(x,a,n) [Page 87]

EXPAND(x,n) [Page 82]

LPAD(x,a,y,n)

LPAD(x,a,y,n) is a string function [Page 81] that inserts the character string y at the start of the character string [Page 16] x as often as specified by the parameter a. Leading and subsequent blanks in the character string x are truncated. The optional parameter n defines the maximum total length of the character string created.

The result of the parameter a must be a positive integer.

The optional parameter n must be greater than or equal to the total LENGTH [Page 76](x)+a*LENGTH(y).

	Result of the LPAD(x,a,y,n) function
LPAD(x,a,y) x must identify a CHAR or VARCHAR column.	The maximum length of the character string is the length of the character string x.
x or a is the NULL value	NULL value [Page 15]
a is the special NULL value [Page 15]	Error Message

Example table: customer [Page 111]

Creating bar charts: LPAD inserts asterisks in front of the first parameter (in this case, a blank). This is done according to the number equal to account divided by 100.

```
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	*

See also:

RPAD(x,a,y,n) [Page 88]

LTRIM(x,y)

LTRIM(x,y) is a string function [Page 81] that removes all of the characters specified in the character string y from the start of the character string [Page 16] x. The result of LTRIM(x,y), therefore, starts with the first character that was not specified in y.

	Result of the LTRIM(x,y) function
LTRIM(x)	Only blanks (code attribute ASCII, EBCDIC (code tables [Page 18])) or binary zeros (code attribute BYTE) are removed from x.
x is NULL value	NULL value [Page 15]

See also:

TRIM(x,y) [Page 90]

RTRIM(x,y) [Page 88]

MAPCHAR(x,n,i)

MAPCHAR(x,n,i) **is a** string function [Page 81] that converts country-specific letters to a different format (such as German umlauts, French letters with a grave accent). These letters are located in the ASCII code [Page 18] and EBCDIC code [Page 20] at positions that can seldom be used for sorting purposes. The MAPCHAR(x,n,i) function cannot be used for UNICODE databases.

MAPCHAR(x,n,i) uses the MapChar set with the name i (mapchar set name [Page 42]) to convert the character string x. If you do not specify a MapChar set name, the MapChar set with the name DEFAULTMAP is used.

The parameter n is optional and specifies the maximum length of the result.

	Result of the MAPCHAR(x,n,i) function
MAPCHAR(x,i)	MAPCHAR(x,n,i), whereby n is the length of the character string x
MAPCHAR(x,i) x is CHAR or VARCHAR column	MAPCHAR(x,n,i), whereby n is the length of the column x
MAPCHAR(x)	MAPCHAR(x,DEFAULTMAP)
x is a NULL value	NULL value [Page 15]

The function MAPCHAR enables data to be sorted correctly, e.g. if "ü" is to be treated for as "UE" sorting purposes. The MAPCHAR SET with the name DEFAULTMAP is used.

```
SELECT...,MAPCHAR(<column name>) sort,...FROM...ORDER BY
sort
```

REPLACE(x,y,z)

REPLACE(x,y,z) is a string function [Page 81] that replaces the character string y in the character string [Page 16] x with the character string z.

	Result of the REPLACE(x,y,z) function
REPLACE(x,y)	The character string y in x is deleted
x is a NULL value	NULL value [Page 15]
y is NULL value	x remains unchanged
z is NULL value	The character string y in x is deleted

Example table: hotel [Page 112]

The abbreviated notation for Street (St.) is to be written out in full (Street) for the sake of uniformity.

```
SELECT hno, zip, city, REPLACE (address,' t.' ,' treet' )
address
FROM hotel where address like '%tr%'
```

HNO	ZIP	CITY	ADDRESS
10	89477	Detroit	155 Beechwood Street
20	86159	Cincinnati	1499 Grove Street
80	20251	Los Angeles	12 Barnard Street
100	50668	Dallas	1980 34th Street
110	81245	New York	111 78th Street

See also:

TRANSLATE(x,y,z) [Page 90]

RFILL(x,a,n)

RFILL(x,a,n) is a string function [Page 81] that inserts the character string a at the end of the character string [Page 16] x as often as required until the character string has reached the length n. If the character string a cannot be completely inserted without exceeding the specified total length, only the part that is needed to reach the total length is inserted.

	Result of the RFILL(x,a,n) function
RFILL(x,a) x must identify a CHAR or VARCHAR column.	The column x is filled with the characters of a until its maximum length is reached.
x is NULL value	NULL value [Page 15]
a or n is the NULL value	Error message

See also:

LFILL(x,a,n) [Page 84]

EXPAND(x,n) [Page 82]

RPAD(x,a,y,n)

RPAD(x,a,y,n) is a string function [Page 81] that inserts the character string y at the end of the character string [Page 16] x as often as specified by the parameter a. Leading and subsequent blanks in the character string x are truncated. The optional parameter n defines the maximum total length of the character string created.

The result of the parameter a must be a positive integer.

The optional parameter n must be greater than or equal to the total LENGTH [Page 76](x)+a*LENGTH(y).

	Result of the RPAD(x,a,y,n) function
RPAD(x,a,y) x must identify a CHAR or VARCHAR column.	The maximum length of the character string is the length of the character string x.
x or a is the NULL value	NULL value [Page 15]
a is the special NULL value [Page 15]	Error message

See also:

LPAD(x,a,y,n) [Page 85]

RTRIM(x,y)

RTRIM(x,y) is a string function that removes the blanks (code attribute ASCII, EBCDIC (code tables [Page 18])) or binary zeros (code attribute BYTE) from the end of the character string [Page 16] x and then all of the characters specified in the character string y. The result of RTRIM(x,y), therefore, ends with the last character that was not specified in y.

	Result of the RTRIM(x,y) function
RTRIM(x)	Only blanks (code attribute ASCII, EBCDIC or binary zeros (code attribute BYTE) are removed from x.
x is NULL value	NULL value [Page 15]

See also:

TRIM(x,y) [Page 90]

LTRIM(x,y) [Page 86]

SOUNDEX(x)

SOUNDEX(x) is a string function [Page 81] that converts a character string [Page 16] x to a format that is generated by the SOUNDEX algorithm.

This representation can be used if the user does not know the exact spelling of the search term.

	Result of the SOUNDEX(x) function
SOUNDEX(x)	The SOUNDEX algorithm is used. The result is a value with the data type CHAR(4).

x is NULL value	NULL value [Page 15]

The SOUNDS predicate [Page 70] is often applied to a column x.

Since inversions cannot be used here, it is advisable for performance reasons to define an additional table column x1 with the data type CHAR(4), in which the result of SOUNDEX(x) is then inserted.

The requests should then refer to x1:
Instead of `x SOUNDS LIKE <string literal>`,
use `x1= SOUNDEX(<string literal>)`

SUBSTR(x,a,b)

SUBSTR(x,a,b) is a string function [Page 81] that outputs part of x (character string [Page 16] with length n).

	Result of the SUBSTR(x,a,b) function
SUBSTR(x,a,b)	Part of the character string x that starts at the a^{th} character and is b characters long.
SUBSTR(x,a)	SUBSTR(x,a,n-a+1) supplies all of the characters in the character string x from the a^{th} character to the last (n^{th}) character.
b is an unsigned integer [Page 34]	SUBSTR(x,a,b) b can also have a value that is greater than (n-a+1).
b is not an unsigned integer	SUBSTR(x,a,b) b must not be greater than (n-a+1).
b>(n-a+1)	SUBSTR(x,a) As many blanks (code attribute ASCII, EBCDIC) or binary zeros (code attribute BYTE) are appended to the end of this result as are needed to give the result the length b.
x, a or b is the NULL value	NULL value [Page 15]

Example table: customer [Page 111]

The SUBSTR function is used to reduce the firstname to one letter, add a period and a blank, and then 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
M. Porter	Los Angeles
S. Peters	Los Angeles
P. Brown	Hollywood
M. Porter	New York
G. Howe	New York

F. Randolph	Los Angeles
J. Peters	Los Angeles
S. Brown	Los Angeles
A. Jackson	Los Angeles
T. Adams	Los Angeles
M. Griffith	New York
I. Braun	Los Angeles

TRANSLATE(x,y,z)

TRANSLATE(x,y,z) is a string function [Page 81] that replaces the i^{th} character of the character string [Page 16] y with the i^{th} character of the character string z in the character string x. The character strings y and z must have the same length.

	Result of the TRANSLATE(x,y,z) function
x is a NULL value	NULL value [Page 15]
y is NULL value	x remains unchanged

Example table: customer [Page 111]

Each occurrence of the i^{th} letter in the first character string is replaced by the i^{th} letter in the second one.

```
SELECT name, TRANSLATE (name,'or','es') new_name
FROM customer WHERE firstname IS NOT NULL AND city = 'Los
Angeles'
```

NAME	NEW_NAME
Porter	Pestes
Peters	Petess
Randolph	Sandelph
Peters	Petess
Brown	Bsewn
Jackson	Jacksen
Adams	Adams

See also:

REPLACE(x,y,z) [Page 86]

TRIM(x,y)

TRIM(x,y) is a string function [Page 81] that removes all of the characters specified in the character string y from the start of the character string [Page 16] x. The result of TRIM(x,y), therefore, starts with the first character that was not specified in y.

TRIM(x,y) also removes the blanks (code attribute ASCII, EBCDIC (code tables [Page 18])) or binary zeros (code attribute BYTE) from the end of the character string and then all of the characters specified in the character string y. The result of TRIM(x,y), therefore, ends with the last character that was not specified in y.

	Result of the TRIM(x,y) function
TRIM(x)	Only blanks (code attribute ASCII, EBCDIC or binary zeros (code attribute BYTE) are removed from x.
x is a NULL value	NULL value [Page 15]

Example table: customer [Page 111]

```
SELECT name, TRIM(CHR(account)) & ' EURO' account
FROM customer WHERE account > 0.00
```

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

See also:

LTRIM(x,y) [Page 86]

RTRIM(x,y) [Page 88]

UPPER/LOWER(x)

UPPER(x) and LOWER(x) are string functions [Page 81] that convert a character string [Page 16] to uppercase/lowercase letters.

	Result of the Function UPPER(x) or LOWER(x)
x is a NULL value	NULL value [Page 15]

Example table: hotel [Page 112]

Refining searches by specifying uppercase/lowercase letters

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

HNO	NAME	ZIP	CITY	ADDRESS
30	Regency	48153	Portland	477 17th Avenue

Concatenation (concatenation)

A concatenation (`concatenation`) x||y or x&y is a string function [Page 81] that supplies the following results for x (character string [Page 16] with the length n) and y (character string with the length m):

	Result of the concatenation
x||y or x&y	x and y are concatenated to a character string with the length n+m. If a character string originates from a column, its length is determined without any consideration of trailing blanks (code attribute [Page 17]. ASCII, EBCDIC, UNICODE) or binary zeros (code attribute BYTE).
x or y is the NULL value	NULL value [Page 15]

Columns with the same code attribute can be concatenated.

Columns with different code attributes (ASCII, EBCDIC and UNICODE) can be concatenated together as well as with date value [Page 17]s, time value [Page 17]s, and timestamp value [Page 17]s.

Example table: customer [Page 111]

```
SELECT name, zip & ' - ' & city address FROM customer
WHERE city = 'New York'
```

NAME	ADDRESS
Porter	80335 - New York
Porter	70596 - New York
Howe	81737 - New York
Griffith	81739 - New York

Date function

A date function is a function [Page 73] that is applied to date or timestamp values or supplies a date or timestamp value as a result.

Syntax

```
<date_function> ::=
  ADDDATE    ( <date_or_timestamp_expression>, <expression> )
| SUBDATE    ( <date_or_timestamp_expression>, <expression> )
| DATEDIFF   ( <date_or_timestamp_expression>,
<date_or_timestamp_expression> )
| DAYOFWEEK  ( <date_or_timestamp_expression> )
| WEEKOFYEAR ( <date_or_timestamp_expression> )
| DAYOFMONTH ( <date_or_timestamp_expression> )
| DAYOFYEAR  ( <date_or_timestamp_expression> )
| MAKEDATE   ( <expression>, <expression> )
| DAYNAME    ( <date_or_timestamp_expression> )
| MONTHNAME  ( <date_or_timestamp_expression> )
```

date_or_timestamp_expression [Page 95], expression [Page 52]

ADDDATE(t,a)/SUBDATE(t,a) [Page 93], DATEDIFF(t,s) [Page 93], DAYOFWEEK(t), WEEKOFYEAR(t), DAYOFMONTH(t), DAYOFYEAR(t) [Page 94], MAKEDATE(a,b) [Page 94], DAYNAME(t), MONTHNAME(t) [Page 94]

Explanation

Although the Gregorian calendar was not introduced until 1582, it can also be applied to date functions that use dates prior to that year. This means that every year is assumed to have either 365 or 366 days.

A variety of date and time formats [Page 50] (ISO, USA, EUR, JIS, INTERNAL) are available for processing date and time values.

ADDDATE/SUBDATE(t,a)

ADDDATE(t,a) and SUBDATE(t,a) are date functions [Page 92] that calculate a date in the future or past.

t: date_or_timestamp_expression [Page 95]

a: numeric value that represents the number of days. Any decimal places in a are truncated if necessary.

	Result of the ADDDATE(t,a)/SUBDATE(t,a) function
Addition of a to t/ subtraction of a from t	Date value [Page 17] or Time stamp value [Page 17]
t or a is NULL value	NULL value [Page 15]
a is special NULL value [Page 15]	Error Message

Example table: reservation [Page 114]

Increasing a reservation date by two days

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

ARRIVAL	ARRIVAL2	RNO
01/02/2002	03/02/2002	130

DATEDIFF(t,s)

DATEDIFF(t,s) is a date function [Page 92] that calculates the number of days between a start and end date.

t and s: date_or_timestamp_expression [Page 95]

	Result of DATEDIFF(t,s) function
Positive difference between t and s	Numeric value (number of days)
t or s are timestamp values	Only the dates [Page 17] in the timestamp value [Page 17] are used to calculate DATEDIFF(t,s).
t or s is NULL value	NULL value [Page 15]

Example table: reservation [Page 114]

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

ARRIVAL	DEPARTURE	DIFFERENCE	RNO
01/02/2002	03/02/2002	2	130

DAYNAME/MONTHNAME(t)

DAYNAME(t) and MONTHNAME(t) are date functions [Page 92] that supply the weekday or month of the specified day.

t: date_or_timestamp_expression [Page 95]

	Result of the function
DAYNAME(t) or MONTHNAME(t)	Character string [Page 16] that supplies the name of a weekday (Sunday to Saturday) or month (January to December).
t is NULL value	NULL value [Page 15]

DAYOFWEEK/WEEKOFYEAR/DAYOFMONTH/DAYOFYEAR(t)

DAYOFWEEK(t)/WEEKOFYEAR(t)/DAYOFMONTH(t)/DAYOFYEAR(t) are date functions [Page 92] that calculate the following:

t: date_or_timestamp_expression [Page 95]

	Result of the function
DAYOFWEEK(t)	Numeric value between 1 and 7 (weekday) The first day is Monday, the second Tuesday, etc.
WEEKOFYEAR(t)	Numeric value between 1 and 53 (calendar week of the specified day)
DAYOFMONTH(t)	Numeric value between 1 and 31 (day of the month of the specified day)
DAYOFYEAR(t)	Numeric value between 1 and 366 (day of the year of the specified day)
t is NULL value	NULL value [Page 15]

MAKEDATE(a,b)

MAKEDATE(a,b) is a date function [Page 92] that supplies a date value [Page 17] from a year and day value.

The values a and b (expression [Page 52]) in MAKEDATE must supply numeric values.
The following restrictions also apply: a>=0 (a represents a year value)

b>0 (b represents a day value)
Decimal places in a and b are truncated if necessary.

	Result of the MAKEDATE(a,b) function
a or b is the NULL value	NULL value [Page 15]
a or b is the special NULL value [Page 15]	Error message

`MAKEDATE(1999,49)` in the INTERNAL date format [Page 50] outputs '19990218'

date_or_timestamp_expression

When used in a function, the `date_or_timestamp_expression` argument must supply a date value [Page 17], a timestamp value [Page 17], or an alphanumeric value that matches the current date or timestamp format [Page 50].

Time function

A time function is a function [Page 73] that is applied to time or timestamp values or supplies a time or timestamp value as a result.

Syntax

```
<time_function> ::=
  ADDTIME     ( <time_or_timestamp_expression>, <time_expression> )
| SUBTIME     ( <time_or_timestamp_expression>, <time_expression> )
| TIMEDIFF    ( <time_or_timestamp_expression>,
<time_or_timestamp_expression> )
| MAKETIME    ( <hours>, <minutes>, <seconds> )
```

time_or_timestamp_expression [Page 97], time_expression [Page 96], hours, minutes, seconds [Page 96]

ADDTIME/SUBTIME(t,a) [Page 95], TIMEDIFF(t,s) [Page 96], MAKETIME(h,m,s) [Page 96]

A variety of date and time formats [Page 50] (ISO, USA, EUR, JIS, INTERNAL) are available for processing date and time values.

ADDTIME/SUBTIME(t,a)

ADDTIME(t,a) and SUBTIME(t,a) are time functions [Page 95] that calculate a time/timestamp value in the past or future.

t: time_or_timestamp_expression [Page 97]
a: time_expression [Page 96]

	Result of the ADDTIME(t,a)/SUBTIME(t,a) function
Addition of a to t/ subtraction of a from t	Time value [Page 17] or time stamp value [Page 17]
SUBTIME(t,a) and t and a are time values	a must be less than t, otherwise an error is output

t or a is NULL value	NULL value [Page 15]

MAKETIME(h,m,s)

MAKETIME(h,m,s) is a time function [Page 95] that calculates a time value [Page 17] from the total number of hours, minutes, and seconds [Page 96].

	Result of the MAKETIME(h,m,s) function
h,m,s is NULL value	NULL value [Page 15]
h,m,s is special NULL value [Page 15]	Error message

TIMEDIFF(t,s)

TIMEDIFF(t,s) is a time function [Page 95] that calculates the time value between a start and end time.

t and s: time_or_timestamp_expression [Page 97]
Both arguments must have the same data type, i.e. they must be either a time value [Page 17] or a timestamp value [Page 17].

	Result of the TIMEDIFF(t,s) function
Positive difference between t and s	Time value
t or s are timestamp values for alphanumeric values that match the current timestamp format.	The dates [Page 17] contained in the timestamp value are used to calculate TIMEDIFF(t,s).
Difference of more than 9999 hours	Number of hours modulo 10000
t or s is NULL value	NULL value [Page 15]

hours/minutes/seconds

When used in a function, each of these arguments (`hours`, `minutes` or `seconds`) must be an integer greater than or equal to 0.

Decimal places are truncated.

Time expression

When used in a function, the argument `time_expression` must supply a time value [Page 17] or an alphanumeric value that is in the current time format [Page 50].

Time or timestamp expression

When used in a function, the argument `time_or_timestamp_expression` must supply a time value [Page 17], timestamp value [Page 17], or an alphanumeric value that matches the current time or timestamp format [Page 50].

Extraction function

An extraction function is a function [Page 73] that extracts parts of a date value [Page 17], time value [Page 17], or timestamp value [Page 17] or that calculates a date, time, or timestamp value.

Syntax

```
<extraction_function> ::=
  YEAR         ( <date_or_timestamp_expression> )
| MONTH        ( <date_or_timestamp_expression> )
| DAY          ( <date_or_timestamp_expression> )
| HOUR         ( <time_or_timestamp_expression> )
| MINUTE       ( <time_or_timestamp_expression> )
| SECOND       ( <time_or_timestamp_expression> )
| MICROSECOND  ( <expression> )
| TIMESTAMP    ( <expression>[, <expression> ] )
| DATE         ( <expression> )
| TIME         ( <expression> )
```

date_or_timestamp_expression [Page 95], time_or_timestamp_expression [Page 97], expression [Page 52]

YEAR(t), MONTH(t), DAY(t) [Page 99], HOUR(t), MINUTE(t), SECOND(t) [Page 98], MICROSECOND(a) [Page 98], TIMESTAMP(a,b) [Page 98], DATE(a) [Page 97], TIME(a) [Page 98]

A variety of date and time formats [Page 50] (ISO, USA, EUR, JIS, INTERNAL) are available for processing date and time values.

DATE(a)

DATE(a) is a function (extraction [Page 97]) that calculates a date value [Page 17].

	Result of DATE(a) function
a is a date value or an alphanumeric value that matches the current date format	This date value
a is an alphanumeric value that does not match the current date format	Error message
a is a timestamp value [Page 17] or an alphanumeric value that matches the current timestamp format	The date value that is part of the timestamp
a is a fixed point or floating point number [Page 16]	Date value that is equal to the x^{th} day after December 31, 0000 (x=TRUNC(a) [Page 79])
a is NULL value	NULL value [Page 15]

a is special NULL value [Page 15]	Error message

HOUR/MINUTE/SECOND(t)

HOUR(t), MINUTE(t), and SECOND(t) are functions (extraction [Page 97]) that extract the hours, minutes, or seconds from the specified time or timestamp value.

t: time_or_timestamp_expression [Page 97]

	Result of the function
HOUR(t), MINUTE(t), or SECOND(t)	Numeric value (hours, minutes, and seconds)
t is NULL value	NULL value [Page 15]

MICROSECOND(a)

MICROSECOND(a) is a function (extraction [Page 97]) that extracts the microseconds from a.

The value a must be a timestamp value [Page 17] or supply an alphanumeric value that matches the current timestamp format.

	Result of MICROSECOND(a) function
MICROSECOND(a)	Numeric value (microseconds)
a is NULL value	NULL value [Page 15]

TIME(a)

TIME(a) is a function (extraction [Page 97]) that calculates a time value [Page 17].

	Result of TIME(a) function
a is a time value or an alphanumeric value that matches the current time format	This time value
a is an alphanumeric value that does not match the current time format	Error message
a is a timestamp value [Page 17] or an alphanumeric value that matches the current timestamp format	The time value that is part of the timestamp
a is NULL value	NULL value [Page 15]

TIMESTAMP(a,b)

TIMESTAMP(a,b) is a function (extraction [Page 97]) that calculates a timestamp value [Page 17] comprising a date value [Page 17], time value [Page 17], and 0 microseconds.

	Result of TIMESTAMP(a,b) function
TIMESTAMP(a)	a must be a timestamp value or an alphanumeric value that matches the current timestamp format The result is this timestamp value.
TIMESTAMP(a,b)	a must be a date value and b a time value (or alphanumeric value that matches the current format for date and time values). The result is a timestamp value calculated from a date value, time value, and 0 microseconds.
a or b is the NULL value	NULL value [Page 15]

YEAR/MONTH/DAY(t)

YEAR(t), MONTH(t), and DAY(t) are functions (extraction [Page 97]) that extract the year, month, and day from the specified date or timestamp value.

t: date_or_timestamp_expression [Page 95]

	Result of the function
YEAR(t), MONTH(t), or DAY(t)	Numeric value (year, month, day)
t is NULL value	NULL value [Page 15]

Special Function (special_function)

There are certain special functions [Page 73] (`special_function`) that are not restricted to specific data types.

Syntax

```
<special_function> ::=
  VALUE    (<expression>,<expression>,...)
| GREATEST (<expression>,<expression>,...)
| LEAST    (<expression>,<expression>,...)
| DECODE   (<check_expression>,<search_and_result_spec>,...[,<default
_expression>])
| <case_function>

<check_expression> ::= <expression>
<search_and_result_spec> ::= <search_expression>,<result_expression>
<default_expression> ::= <expression>
<result_expression> ::= <expression>
<case_function> ::= <simple_case_function>|<searched_case_function>
```

expression [Page 52]

VALUE(x,y,...) [Page 101], GREATEST(x,y,...), LEAST(x,y,...) [Page 100], DECODE(x,y,...,z) [Page 100], simple_case_function [Page 102], searched_case_function [Page 101]

DECODE(x,y(i),...,z)

DECODE(x,y(i),...,z) is a special function [Page 99] that decodes expressions [Page 52] in accordance with their values.

x	check_expression	Expression (expression) for which a comparison is to be carrie the values in y(i)
y(i)	search_and_result_spec	<search_and_result_spec> ::= <search_expression> <result_expression> (y(i)=u(i),v(i), i=1,...) Combination of the comparison value u(i) and the value v(i) that replace this comparison value
z	default_expression	Optional default value
u(i)	search_expression	Comparison value that is to be replaced by v(i) if it matches x
v(i)	result_expression	Value that is to be replace u(i)

The data types of x and u(i) must be comparable. The data types of v(i) and z must be comparable. The data types of u(i) and v(i) do not have to be comparable.

DECODE compares the values of x with the values u(i) consecutively. If a match is found, the result of DECODE is the value v(i) in the combination y(i)=u(i),v(i).

A match is present if x and u(i) are NULL values. The comparison of the special NULL value with any other value never results in a match.

If a match is not found, DECODE supplies the result of z. If z is not specified, the NULL value is the result of DECODE.

Example table: room [Page 113]

The room type identifiers are to be replaced in the output by an identifier declared in the DECODE function.

```
SELECT hno, price, DECODE (roomtype,
'SINGLE', 1, 'DOUBLE', 2, 'SUITE', 3) room_code
FROM room
```

GREATEST/LEAST(x,y,...)

GREATEST(x,y,...) or LEAST(x,y) is a special function [Page 99] that calculates the maximum or minimum value of all arguments.

The functions can be applied to any data type. The data types of the individual arguments must be comparable.

	Result of the function
At least one argument is the NULL value	NULL value [Page 15]
At least one argument is the special NULL value	Special NULL value [Page 15]

VALUE(x,y,...)

VALUE(x,y,...) is a special function [Page 99] that can be used to replace NULL values [Page 15] with a non-NULL value.

The arguments of the VALUE function must be comparable. The arguments are evaluated one after the other in the specified order.

	Result of the VALUE(x,y) function
One of the arguments is a non-NULL value	The first non-NULL value that occurs
Each argument is a special NULL value [Page 15]	Special NULL value
Each argument is a NULL value	NULL value

Example table: customer [Page 111]

The title does not occur in the output list. The word `company` is to be output for companies in the `firstname` column instead of a NULL value.

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

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

General CASE Function (searched_case_function)

The general CASE function (`searched_case_function`) is a special function [Page 99] that analyzes a quantity of search conditions to determine a `result_expression`.

Syntax

```
<searched_case_function> ::=
CASE
  WHEN <search_condition> THEN <result_expression>
    [...]
  [ELSE <default_expression>])
END

<result_expression> ::= <expression>
<default_expression> ::= <expression>
```

search_condition [Page 70], expression [Page 52]

Explanation

u(i)	search_condition	Search conditions u(i). All search conditions without ROWNO Predicate [Page 69] are permissible.
v(i)	result_expression	Value v(i) that is to be accepted, if the search condition u(i) is true
z	default_expression	Value z, optional default value

CASE checks the search conditions u(i) in succession. As soon as a search condition that is true is found, the result of the general CASE function is the value v(i) that belongs to u(i).

The data types of v(i) and z must be comparable.

If no search condition that is true is found, CASE returns the result of z. If z is not specified, the NULL value [Page 15] is the result of CASE.

```
CASE
  WHEN price IS NULL THEN 'Not yet priced'
  WHEN price < 10 THEN 'Very Reasonable Title'
  WHEN price >= 10 and price < 20 THEN 'Coffee Table Title'
  ELSE 'Expensive book!'
END
```

See also:

Simple CASE function (simple_case_function) [Page 102]

Simple CASE Function (simple_case_function)

The simple CASE function (simple_case_function) is a special function [Page 99] that analyzes a quantity of simple expressions to determine a (result_expression [Page 52]).

Syntax

```
<simple_case_function> ::=
CASE <check_expression>,
        WHEN <search_expression> THEN <result_expression>
            [...]
        [ELSE <default_expression>])
END

<check_expression> | <search_expression> | <result_expression> |
<default_expression> ::= <expression>
```

expression [Page 52]

Explanation

x	check_expression	Expression x for which a comparison is to be carried out with the comparison values u(i)
u(i)	search_expression	Comparison value u(i). The function delivers v(i) as the result at the first match of u(i) with x.
v(i)	result_expression	Value v(i) that is to be accepted, if u(i) matches x
z	default_expression	Value z, optional default value

CASE compares the values of x with the values u(i) consecutively. If a match is determined, the result of the simple CASE function is the value v(i) associated with u(i).

The data types of x and u(i) must be comparable. The data types of v(i) and z must be comparable. The data types of u(i) and v(i) do not have to be comparable.

A match is present if x and u(i) are NULL values. The comparison of the special NULL value with any other value never results in a match.

If a match is not found, CASE supplies the result of z. If z is not specified, the NULL value [Page 15] is the result of CASE.

See also:

General CASE Function (searched_case_function) [Page 101]

Conversion Function (conversion_function)

A conversion function is a function [Page 73] that converts a value from one data type to another.

Syntax

```
<conversion_function> ::=
  NUM    ( <expression> )
| CHR    ( <expression>[, <unsigned_integer> ] )
| HEX    ( <expression> )
| HEXTORAW ( <expression> )
| CHAR   ( <expression>[, <datetimeformat> ] )
```

expression [Page 52], unsigned_integer [Page 34], datetimeformat [Page 50]

NUM(a) [Page 105], CHR(a,n) [Page 104], HEX(a) [Page 104], HEXTORAW(a) [Page 104], CHAR(a,t) [Page 103]

CHAR(a,t)

CHAR(a,t) is a function (conversion function [Page 103]) that converts the date values [Page 17], time values [Page 17], or timestamp values [Page 17] to a character string with the format for date values, time values, or timestamp values [Page 50] specified in t.

	Result of the CHAR(a,t) function
CHAR(a)	The current date and time format is used for the value a.
a is the NULL value	NULL value [Page 15]

CHR(a,n)

CHR(a,n) is a function (conversion function [Page 103]) that converts numbers into character strings.

The CHR function can only be applied to numeric values, character strings [Page 16], and boolean values [Page 17].

	Result of the CHR(a,n) function
a is a numeric value	Character string that matches the CHAR representation of the numeric value a. The code attribute of the character string corresponds to the code type of the computer.
a is character string	Identical character string
a is a boolean value	T, if a=TRUE F, if a=FALSE
a is not a numeric value, a character string, or a boolean value	Error message
CHR(a,n), n>=1	Output with the length attribute n
CHR(a)	The length attribute n is calculated as a function of the data type and the length of a.
CHR(a) and a if of the data type FLOAT(p)	If p=1, then n=6 If p>1, then n=p+6
CHR(a) and a is of the data type FIXED(p,s)	If p=s, then n=p+s If s=0, then n=p+1
a is the NULL value	NULL value [Page 15]
a is the special NULL value [Page 15]	Error message

HEX(a)

HEX(a) is a function (conversion function [Page 103]) that converts the argument a to hexadecimal notation. The function can be applied to any data type.

	Result of the HEX(a) function
a is the NULL value	NULL value [Page 15]
a is the special NULL value [Page 15]	Error message

HEXTORAW(a)

HEXTORAW(a) is a function (conversion function [Page 103]). Arguments of the function must be hexadecimal characters (hex_digit [Page 31]). The argument a is converted to a character string with the code attribute [Page 17] BYTE that contains the corresponding

characters. During this conversion, two hexadecimal characters are converted to one character.

	Result of the HEXTORAW(a) Function
a is the NULL value	NULL value [Page 15]
a is the special NULL value [Page 15]	Error message

NUM(a)

NUM(a) is a function (conversion function [Page 103]) that converts the argument a to a numeric value.

NUM can be applied to character string [Page 16]s with the code attributes ASCII or EBCDIC (see code tables [Page 18]), date value [Page 17]s, time value [Page 17]s, time stamp value [Page 17]s, and to numeric and boolean values (BOOLEAN [Page 17]).

	Result of the NUM(a) function
a is a character string and can be interpreted as a numeric value	Corresponding numeric value
a is a numeric value	Identical numeric value (unchanged)
a is a boolean value	1, if a=TRUE 0, if a=FALSE
a is a character string that cannot be interpreted as a numeric value; a is a character string that does not have the code attribute ASCII or EBCDIC; a is neither a numeric nor a Boolean value.	Error message
a is a character string that can be interpreted as a numeric value outside the range `-9.99999999999999999999999999999999999E+62, 9.99999999999999999999999999999999999E+62`	Special NULL value [Page 15]
a is the NULL value	NULL value [Page 15]
a is the special NULL value	Special NULL value

Set Function (set_function_spec)

There is a series of functions that can be applied to a set of values (rows) as an argument and supply a result. These functions are referred to as set functions (`set function spec`).

Syntax

```
<set_function_spec> ::= COUNT (*) | <distinct_function> |
<all_function>
```

COUNT (*) [Page 108], distinct_function [Page 106], all_function [Page 107]

Explanation

Set functions operate across groups of values but only return one value. The result comprises one row. If a set function is used in a statement, a similar function must also be applied to each of the other columns in the request. This, however, does not apply to columns that were grouped using GROUP BY. In this case, the value of the set function can be defined for each group.

The argument of a DISTINCT function or an ALL function is a result table [Page 23] or a group (the result table can be grouped using a GROUP condition).

With the exception of the COUNT(*) function, no NULL value [Page 15]s are included in the calculation.

No locks are set for certain set functions, irrespective of the isolation level specified when the user connected to the database.

Example table: customer [Page 111]

```
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'
```

SUM_ACCOUNT	MIN_ACCOUNT	AVG_ACCOUNT	MAX_ACCOUNT	NUMBER
-164.17	-4167.79	-20.52	3770.50	8

DISTINCT Function (distinct_function)

The DISTINCT function (`distinct_function`) is a set function [Page 105] that removes duplicated values and all NULLvalue [Page 15]s.

Syntax

```
<distinct function>::= <set function name> ( DISTINCT <expression> )
```

set_function_name [Page 108], expression [Page 52]

Explanation

The argument of a DISTINCT function is a set of values that is calculated as follows:

1. A result table [Page 23] or group (the result table can be grouped with a GROUP condition) is formed.
2. The expression is applied to each row in this result table or group.
   The expression must not contain a set function.
3. All of the NULL values and duplicated values are removed (DISTINCT). Special NULL value [Page 15]s are not removed and two special NULL values are considered identical.

The DISTINCT function is executed taking into account the relevant set function name for this set of values.

	Result of the DISTINCT function

Set of values is empty and the DISTINCT function is applied to the entire result table	The set functions AVG, MAX, MIN, STDDEV, SUM, VARIANCE supply the NULL value as their result. The set function COUNT supplies the value 0.
There is no group to which the DISTINCT function can be applied.	The result is an empty table.
The set of values contains at least one special NULL value.	Special NULL value

Example table: customer [Page 111]

In how many cities do the customers live?

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

number_cities
4

ALL function

The ALL function is a set function [Page 105] that removes the NULL value [Page 15]s.

Syntax

```
<all_function>::= <set_function_name> ( [ALL] <expression> )
```

set_function_name [Page 108], expression [Page 52]

Explanation

The argument of an ALL function is a set of values that is calculated as follows:

1. A result table [Page 23] or group (the result table can be grouped with a GROUP condition) is formed.
2. The expression is applied to each row in this result table or group.
   The expression must not contain a set function.
3. All NULL values are removed. Special NULL value [Page 15]s are not removed and two special NULL values are considered identical.

The ALL function is executed taking into account the relevant set function name for the set of values.

The result of an ALL function is independent of whether the keyword ALL is specified or not.

	Result of the ALL function
The set of values is empty and the ALL function is applied to the entire result table	The set functions AVG, MAX, MIN, STDDEV, SUM, VARIANCE supply the NULL value as their result. The set function COUNT supplies the value 0.
There is no group to which the ALL function can be applied.	The result is an empty table.
The set of values contains at least one special NULL value.	Special NULL value

Set function name

Set function name is the name of a function that can be specified in a DISTINCT function [Page 106] and an ALL function [Page 107].

Syntax

```
<set_function_name> ::= COUNT | MAX | MIN | SUM | AVG | STDDEV |
VARIANCE
```

COUNT [Page 108], MAX, MIN [Page 109], SUM [Page 109], AVG [Page 108], STDDEV [Page 109], VARIANCE [Page 109]

AVG

AVG is a set function [Page 105].

The result of AVG is the arithmetical mean of the values of the argument.

AVG can only be applied to numeric values. The result has the data type FLOAT(38) [Page 122].

Example table: customer [Page 111]

How many corporate customers are taken into account? What is the average value of their account balance?

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

NUMBER	AVG_ACCOUNT
2	4292.00

COUNT

COUNT is a set function [Page 105].

- `COUNT(*)` supplies the total number of values (rows in a result table or group).
- `COUNT(DISTINCT <expression>)` supplies the total number of different values (number of values in the argument of the DISTINCT function [Page 106]).
- `COUNT(ALL <expression>)` supplies the number of values that differ from the NULL value (number of values in the argument of the ALL function [Page 107])

The result has the data type FIXED(10) [Page 121].

Example table: customer [Page 111]

How many customers are there?

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

number

```
15
```

MAX/MIN

MAX/MIN is a set function [Page 105].

The result of MAX is the largest value of the argument.

The result of MIN is the smallest value of the argument.

STDDEV

STDDEV is a set function [Page 105].

The result of STDDEV is the standard deviation of the values of the argument.

STDDEV can only be applied to numeric values. The result has the data type FLOAT [Page 122](38).

SUM

SUM is a set function [Page 105].

The result of SUM is the sum of the values of the argument.

SUM can only be applied to numeric values. The result has the data type FLOAT [Page 122](38).

VARIANCE

VARIANCE is a set function [Page 105].

The result of VARIANCE is the variance of the values of the argument.

VARIANCE can only be applied to numeric values. The result has the data type FLOAT [Page 122](38).

SQL Statement: Overview

All SQL statements (`sql_statement`) can be embedded in programming languages. For more information, see the precompiler documentation. All SQL statements, with the exception of NEXT STAMP, can be specified interactively.

Comments [Page 111] can be specified for every SQL statement. There are example statements, using the example tables [Page 111], for many SQL statements.

SQL statements for data definition [Page 115]

CREATE TABLE statement	DROP TABLE statement	ALTER TABLE statement RENAME TABLE statement RENAME COLUMN statement EXISTS TABLE statement

CREATE DOMAIN statement	DROP DOMAIN statement	
CREATE SEQUENCE statement	DROP SEQUENCE statement	
CREATE SYNONYM statement	DROP SYNONYM statement	RENAME SYNONYM statement
CREATE VIEW statement	DROP VIEW statement	RENAME VIEW statement
CREATE INDEX statement	DROP INDEX statement	ALTER INDEX statement RENAME INDEX statement
COMMENT ON statement		
CREATE TRIGGER statement	DROP TRIGGER statement	
CREATE DBPROC statement	DROP DBPROC statement	

SQL statements for authorization [Page 160]

CREATE USER statement	DROP USER statement	ALTER USER statement RENAME USER statement GRANT USER statement
CREATE USERGROUP statement	DROP USERGROUP statement	ALTER USERGROUP statement RENAME USERGROUP statement GRANT USERGROUP statement
CREATE ROLE statement	DROP ROLE statement	
ALTER PASSWORD statement	GRANT statement	REVOKE statement

SQL statements for data manipulation [Page 173]

INSERT statement	UPDATE statement	DELETE statement
NEXT STAMP statement	CALL statement	

SQL statements for data query [Page 184]

QUERY statement	SINGLE SELECT statement	EXPLAIN statement
OPEN CURSOR statement	FETCH statement	CLOSE statement

SQL statements for transaction [Page 210] management

CONNECT statement	SET statement	
COMMIT statement	ROLLBACK statement	SUBTRANS statement
LOCK statement	UNLOCK statement	RELEASE statement

SQL statements for statistics [Page 217] management

UPDATE STATISTICS statement	MONITOR statement	

Comment (sql_comment)

A comment (`sql_comment`) can be included for every SQL statement [Page 109].

Syntax

```
<sql_comment> ::= /*<comment text>*/ | --<comment text>
```

Explanation

You can enter any commend text.

If you use the syntax rule `--<comment text>` in a line, all characters entered in this line after `--` are regarded as comments.

```
CREATE TABLE person (cno FIXED(4), first name CHAR(7), last
name CHAR(7), account FIXED(7,2)) /*create table person*/
```

Example Tables

customer [Page 111]

hotel [Page 112]

room [Page 113]

reservation [Page 114]

customer

Create the table **`customer`** and fill it with the values listed below.

Table Structure

```
CREATE TABLE customer
(cno         FIXED(4) PRIMARY KEY CONSTRAINT cno BETWEEN 1 AND 9999,
 title       CHAR(5) CONSTRAINT title IN ('Mr','Mrs','Comp'),
 name        CHAR(10) NOT NULL,
 firstname   CHAR(7),
 zip         CHAR(5) CONSTRAINT zip like '(0-9)(0-9)(0-9)(0-9)(0-9)',
 city        CHAR(12) NOT NULL,
 account     FIXED(7,2) CONSTRAINT account BETWEEN -10000 AND 10000)
```

Content

CNO	TITLE	NAME	FIRSTNAME	ZIP	CITY	ACCOUNT
3000	Mrs	Porter	Jenny	80335	New York	100.00
3100	Comp	DATASOFT	?	50933	Dallas	4813.50
3200	Mr	Porter	Martin	10969	Los Angeles	0.00
3300	Mrs	Peters	Sally	14165	Los Angeles	0.00
3400	Mr	Brown	Peter	40233	Hollywood	0.00
3500	Mr	Porter	Michael	70596	New York	0.00
3600	Mr	Howe	George	81737	New York	-315.40

3700	Mr	Randolph	Frank	22525	Los Angeles	0.00
3800	Mr	Peters	Joseph	10787	Los Angeles	650.00
3900	Mrs	Brown	Susan	13599	Los Angeles	-4167.79
4000	Mr	Jackson	Anthony	10785	Los Angeles	0.00
4100	Mr	Adams	Thomas	13355	Los Angeles	-416.88
4200	Mr	Griffith	Mark	81739	New York	0.00
4300	Comp	TOOLware	?	13629	Los Angeles	3770.50
4400	Mrs	Brown	Rose	40233	Hollywood	440.00

Other example tables [Page 111]

hotel

Create the table **hotel** and fill it with the values listed below.

Table Structure

```
CREATE TABLE hotel
(hno     FIXED(4) PRIMARY KEY CONSTRAINT hno BETWEEN 1 AND 9999,
 name    CHAR(20) NOT NULL,
 zip     CHAR(5) CONSTRAINT zip like '(0-9)(0-9)(0-9)(0-9)(0-9)',
 city    CHAR(12) NOT NULL,
 address CHAR(25) NOT NULL)
```

Content

HNO	NAME	ZIP	CITY	ADDRESS
10	Congress	89477	Detroit	155 Beechwood St.
20	Los Angeles	86159	Cincinatti	1499 Grove Street
30	Regency	48153	Portland	477 17th Avenue
40	Eight Avenue	21109	Los Angeles	112 8th Avenue
50	Lake Michigan	20097	Los Angeles	354 OAK Terrace 43
60	Airport	60313	New Orleans	650 C Parkway
70	Empire State	80805	New York	65 Yellostone Dr.
80	Midtown	20251	Los Angeles	12 Barnard Street
90	Long Beach	45193	Long Beach	200 Yellostone Dr.
100	Dallas	50668	Dallas	1980 34th St.
110	Atlantic	81245	New York	111 78th Street
120	Sunshine	12249	Los Angeles	35 Broadway 77
130	Star	40223	Hollywood	13 Beechwood Place 2
140	River Boat	70469	New York	788 MAIN STREET 14
150	Indian Horse	69117	Santa Clara	16 MAIN STREET

Other example tables [Page 111]

room

Create the table **room** and fill it with the values listed below.

Table Structure

```
CREATE TABLE room
(hno      FIXED(4),
 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,
 PRIMARY KEY(hno,roomtype))
```

The room table is linked to the hotel [Page 112] table via the hotel number.

Content

HNO	ROOMTYPE	MAX_FREE	PRICE
10	DOUBLE	45	200.00
10	SINGLE	20	135.00
20	DOUBLE	13	100.00
20	SINGLE	10	70.00
30	DOUBLE	15	80.00
30	SINGLE	12	45.00
40	DOUBLE	35	140.00
40	SINGLE	20	85.00
50	DOUBLE	230	180.00
50	SINGLE	50	105.00
50	SUITE	12	500.00
60	DOUBLE	39	200.00
60	SINGLE	10	120.00
60	SUITE	20	500.00
70	DOUBLE	11	180.00
70	SINGLE	4	115.00
80	DOUBLE	19	150.00
80	SINGLE	15	90.00
80	SUITE	5	400.00
90	DOUBLE	145	150.00
90	SINGLE	45	90.00
90	SUITE	60	300.00
100	DOUBLE	24	100.00
100	SINGLE	11	60.00
110	DOUBLE	10	130.00
110	SINGLE	2	70.00

120	DOUBLE	78	140.00
120	SINGLE	34	80.00
120	SUITE	55	350.00
130	DOUBLE	300	270.00
130	SINGLE	89	160.00
130	SUITE	100	700.00
140	DOUBLE	9	200.00
140	SINGLE	10	125.00
140	SUITE	78	600.00
150	DOUBLE	115	190.00
150	SINGLE	44	100.00
150	SUITE	6	450.00

Other example tables [Page 111]

reservation

Create the table **reservation** and fill it with the values listed below.

Table Structure

```
CREATE TABLE reservation
(rno       FIXED(4) PRIMARY 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)
```

A logical link between the customer [Page 111], hotel [Page 112], and room [Page 113] tables is established through the `reservation` table.

Content

RNO	CNO	HNO	ROOMTYPE	ARRIVAL	DEPARTURE
100	3000	80	SINGLE	13/11/2001	15/11/2001
110	3000	100	DOUBLE	24/12/2001	06/01/2002
120	3200	50	SUITE	14/11/2001	18/11/2001
130	3900	110	SINGLE	01/02/2002	03/02/2002
140	4300	80	DOUBLE	12/04/2001	30/04/2001
150	3600	70	DOUBLE	14/03/2002	24/03/2002
160	4100	70	SINGLE	12/04/2001	15/04/2001
170	4400	150	SUITE	01/09/2001	03/09/2001
180	3100	120	DOUBLE	23/12/2001	08/01/2002
190	4300	140	DOUBLE	14/11/2001	17/11/2001

Other example tables [Page 111]

Data definition

The following sections contain an introduction to the data definition language (DDL) used by the database system.

SQL statements for data definition

CREATE TABLE statement [Page 115]	DROP TABLE statement [Page 132]	ALTER TABLE statement [Page 133] RENAME TABLE statement [Page 138] RENAME COLUMN statement [Page 139] EXISTS TABLE statement [Page 139]
CREATE DOMAIN statement [Page 140]	DROP DOMAIN statement [Page 140]	
CREATE SEQUENCE statement [Page 140]	DROP SEQUENCE statement [Page 141]	
CREATE SYNONYM statement [Page 142]	DROP SYNONYM statement [Page 142]	RENAME SYNONYM statement [Page 143]
CREATE VIEW statement [Page 143]	DROP VIEW statement [Page 147]	RENAME VIEW statement [Page 147]
CREATE INDEX statement [Page 148]	DROP INDEX statement [Page 149]	ALTER INDEX statement [Page 149] RENAME INDEX statement [Page 149]
COMMENT ON statement [Page 150]		
CREATE TRIGGER statement [Page 158]	DROP TRIGGER statement [Page 159]	
CREATE DBPROC statement [Page 152]	DROP DBPROC statement [Page 158]	

CREATE TABLE Statement (create_table_statement)

A CREATE TABLE statement (`create_table_statement`) defines a base table (see Table [Page 23]).

Syntax

```
<create table statement> ::=
  CREATE TABLE <table name> (<column definition>[,<table description
element>,...])
  [IGNORE ROLLBACK] [<sample definition>]
| CREATE TABLE <table name> [(<table description element>,...)]
  [IGNORE ROLLBACK] [<sample_definition>] AS <query_expression>
```

```
[<duplicates_clause> ]
| CREATE TABLE <table_name> LIKE <table_name> [IGNORE ROLLBACK]

<table_description_element> ::= <column_definition> |
<constraint_definition> | <referential_constraint_definition> |
<key_definition> | <unique_definition>
```

table_name [Page 47], sample_definition [Page 117], query_expression [Page 189], duplicates_clause [Page 176], column_definition [Page 118], constraint_definition [Page 127], referential_constraint_definition [Page 128], key_definition [Page 131], unique_definition [Page 132]

SQL statement for creating a table called `person`:

```
CREATE TABLE person (cno FIXED(4), firstname CHAR(7), name
CHAR(7), account FIXED(7,2))
```

This CREATE TABLE statement comprises the keywords CREATE TABLE followed by the table name and (in parentheses) a list of column names, separated by commas. You can also define other criteria, such as a primary key, or referential integrity conditions.

Other examples: customer [Page 111], hotel [Page 112], reservation [Page 114], room [Page 113]

Explanation

Executing a CREATE TABLE statement causes data that describes the table (or base table) to be stored in the database catalog. This data is called metadata.

A CREATE TABLE statement cannot contain more than one key definition.

The table name must not be identical with the name of an existing table of the current user.

The current user becomes the owner [Page 41] of the new table. In other words, he or she obtains the INSERT, UPDATE, DELETE, and SELECT privileges for this table. If the table is not a temporary table, the owner is also granted the INDEX, REFERENCES, and ALTER privileges.

See also:

Restrictions [Page 219]

Owner of a table

- **The table owner must be specified in front of the table name:** temporary tables are a special type of table. They only exist during a user database session and are deleted with their entire contents afterwards. Temporary tables are identified by the owner TEMP in front of the table name.
  If a table name has an owner other than TEMP, the owner must be identical to the name of the current user and the user must have the status DBA or RESOURCE.
- **The owner of the table is not specified**: the result is the same as if the current user were the owner.

CREATE TABLE … AS <query_expression> …

- If a **QUERY expression is not specified**, the CREATE TABLE statement must contain at least one column definition.
- If a **query expression** is specified, a base table is defined with the same structure as the result table defined by the QUERY expression.
  If column definitions are specified, the column definition may only consist of a column name [Page 46] and the number of column definitions must be equal to the number of columns in the result table generated by the QUERY expression.
  The data_type [Page 119] of the i^{th} column in the base table is identical to that of the i^{th}

column in the result table generated by the QUERY expression.
The result table may also contain LONG columns [Page 16].
If no column definitions are specified, the column names of the result table are used.
The rows of the result table are implicitly inserted in the generated base table. The DUPLICATES clause [Page 176] can be used to determine how key collisions are handled.
The QUERY expression is subject to certain restrictions that also apply to the INSERT statement [Page 173].

LIKE <table_name>

If `LIKE <table_name>` is specified, an empty base table is created which, from the point of view of the current user, has the same structure as the source table, that is, it has all the columns with the same column names and definitions as the source table. This view does not necessarily have to be identical to the actual structure of the source table, since the user may not know all the columns because of privilege limitations.

The specified table [Page 23] must be either a base table, a view table, or a synonym [Page 24]. The user must have at least one privilege for this table.

The current user is the owner of the base table.

If all the key columns of the table specified after LIKE are contained in the base table, they form the key columns in this table. Otherwise, the database system implicitly inserts a key column SYSKEY CHAR(8) BYTE which then represents the key for the base table.

DEFAULT specification [Page 125]s or CONSTRAINT definitions [Page 127] for columns that are copied to the base table also apply to the new base table.

IGNORE ROLLBACK

IGNORE ROLLBACK is optional and can only be specified for temporary tables. Temporary tables with this characteristic are not affected by the transaction mechanism; i.e., changes affecting these tables are not reversed by rolling back a transaction.

SQL statements for changing table properties

Adding, deleting columns, changing data types, changing the CONSTRAINT definition

ALTER TABLE statement [Page 133]

Renaming columns

RENAME COLUMN Statement [Page 139]

Renaming tables

RENAME TABLE statement [Page 138]

SAMPLE definition

A SAMPLE definition defines the number of rows in a table that are to be used when statistics are updated.

Syntax

```
<sample_definition> ::= SAMPLE <unsigned_integer> ROWS
| SAMPLE <unsigned_integer> PERCENT
```

unsigned_integer [Page 34]

Explanation

The database system manages statistics for each base table. These statistics are used to determine the best strategy for executing an SQL statement. The statistics are stored in the catalog by the UPDATE STATISTICS statement [Page 217].

If a SAMPLE definition is specified in an UPDATE STATISTICS statement, it specifies the number of rows in the table that are to be used to calculate the statistics.

If a SAMPLE definition is not specified in an UPDATE STATISTICS statement and if it is not mandatory that all of the rows in the table be used to calculate the statistics, the database system uses the appropriate SAMPLE definition of the CREATE TABLE or ALTER TABLE statement.

The number of rows for which the UPDATE STATISTICS statement is to be executed can be defined by specifying a numeric or percentage value.

- If a SAMPLE definition is specified as a `PERCENT`, the unsigned integer must be between 1 and 100.
- If a SAMPLE definition is not defined, the database system uses the value 20000 ROWS.

SQL statements in which the SAMPLE definition can be used

CREATE TABLE statement [Page 115]

ALTER TABLE statement [Page 133]

UPDATE STATISTICS statement [Page 217]

Column Definition (column_definition)

A column definition defines a column [Page 24] in a table. The name and data type of each column are defined by the column name and data type. The column names must be unique within a base table.

Syntax

```
<column_definition> ::= <column_name> <data_type>
[<column_attributes>]
| <column_name> <domain_name> [<column_attributes>]
```

column_name [Page 46], data_type [Page 119], column_attributes [Page 124], domain_name [Page 41]

Explanation

If the [PRIMARY] KEY column attribute is specified, the CREATE TABLE statement [Page 115] must not contain a key definition [Page 131].

A column definition may only consist of a column name if a QUERY expression is used in the CREATE TABLE statement.

If a **column name and domain name** (the name of a value range) are specified, the domain name must identify an existing domain. The data type and the length of the domain are assigned to the specified column. If the domain has a constraint_definition [Page 127], the effect is the same as if the corresponding CONSTRAINT definition were specified in the column attribute of the column definition.

Columns, which are part of the key, or for which NOT NULL was defined, are called **NOT NULL columns**. A NULL value [Page 15] cannot be inserted in these columns.

- **Mandatory columns**: NOT NULL columns for which a DEFAULT specification [Page 125] has not been declared as a column attribute are called mandatory columns. Whenever rows are inserted, values must be specified for these columns.
- **Optional columns**: columns that are not mandatory are referred to as optional columns. A value does not have to be specified when a row is inserted in these columns. If a DEFAULT specification exists for the column, the default value is entered in the column. If there is no DEFAULT specification, a NULL value is entered in the column.

Memory requirements of a column value as a function of the data type [Page 123]

See also:

Restrictions [Page 219]

Data Type (data_type)

As well as specifying the column name when you define columns [Page 118], you can also specify data types.

Syntax

```
<data_type> ::=
  CHAR[ACTER] [(<unsigned_integer>)] [ASCII | BYTE | EBCDIC |
UNICODE]
| VARCHAR [(<unsigned_integer>)] [ASCII | BYTE | EBCDIC | UNICODE]
| LONG [VARCHAR] [ASCII | BYTE | EBCDIC | UNICODE]
| BOOLEAN
| FIXED (<unsigned_integer> [,<unsigned_integer>])
| FLOAT (<unsigned_integer>)
| INT[EGER] | SMALLINT
| DATE | TIME | TIMESTAMP
```

unsigned_integer [Page 34]

CHAR[ACTER] [Page 120], VARCHAR [Page 120], LONG[VARCHAR] [Page 121], BOOLEAN [Page 121], FIXED [Page 121], FLOAT [Page 122], INT[EGER] [Page 122], SMALLINT [Page 122], DATE [Page 122], TIME [Page 123], TIMESTAMP [Page 123]

Explanation

For the following character strings [Page 16], a code attribute [Page 17] can be entered as part of a column definition (`column_definition`), if required: CHAR[ACTER], VARCHAR, LONG[VARCHAR]

In addition to the data types defined above, the following data types are permitted in a column definition and are mapped as follows to the data types below:

Data Type	Is Mapped To
BINARY(p)	FIXED(p)
DEC[IMAL](p,s)	FIXED(p,s)
DEC[IMAL](p)	FIXED(p)
DEC[IMAL]	FIXED(5)
DOUBLE PRECISION	FLOAT(38)
FLOAT	FLOAT(16)
FLOAT(39..64)	FLOAT(38)
LONG VARCHAR	LONG

NUMERIC(p,s)	FIXED(p,s)
NUMERIC(p)	FIXED(p)
NUMERIC	FIXED(5)
REAL(p)	FLOAT(p)
REAL	FLOAT(16)
SERIAL	FIXED(10) DEFAULT SERIAL [Page 22]
SERIAL(p)	FIXED(10) DEFAULT SERIAL(p)

See also:

Data type [Page 15]

Memory requirements of a column value as a function of the data type [Page 123]

CHAR[ACTER]

Definition

An alphanumerical column is defined. Specification of length attribute n is optional. If no other length attribute is specified, then n=1.

CHAR[ACTER] [(n)]: 0<n<=8000
CHAR[ACTER] [(n)] UNICODE: 0<n<=4000

The database system determines, in accordance with n, whether the values in the column are stored with a fixed or variable length. If you want to store the values in a variable length regardless of the value of n, you must enter a value for VARCHAR [Page 120].

Use

Specification of data type CHAR[ACTER] in column definition [Page 118], with a code attribute [Page 17], if required.

Integration

Data type (data_type) [Page 119]

VARCHAR

Definition

An alphanumerical column is defined. Specification of length attribute n is optional. If no other length attribute is specified, then n=1.

VARCHAR [(n)]: 0<n<=8000
VARCHAR [(n)] UNICODE: 0<n<=4000

Use VARCHAR (n) if the values in the column are to be stored with a variable length, irrespective of n.

Use

Specification of data type VARCHAR in column definition [Page 118], with a code attribute [Page 17], if required.

Integration

Data type (data_type) [Page 119]

LONG[VARCHAR]

Definition

An alphanumeric column is defined with any length (not for temporary tables).

LONG[VARCHAR]: A maximum of 2 GB of characters can be written in a LONG column.
LONG[VARCHAR] UNICODE: A maximum of 2 GB bytes can be written in a LONG column.

You can only give LONG columns [Page 16] NOT NULL or a DEFAULT specification [Page 125] as a column attribute [Page 124].

Use

Specification of data type LONG in column definition [Page 118], with a code attribute [Page 17], if required.

LONG columns can be used in the following SQL statements:
INSERT statement [Page 173], UPDATE statement [Page 178], NULL predicate [Page 67], and in selected columns [Page 193].

Integration

Data type (data_type) [Page 119]

BOOLEAN

Definition

BOOLEAN: The column is given the data type BOOLEAN [Page 17].

Use

Specifying the data type BOOLEAN when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

FIXED

Definition

FIXED (p,s): A column with a fixed point number [Page 16] with precision p and with s number of decimal places (0<p<=38, s<=p). If no s is specified, it is assumed that the decimal places are 0.

Use

Specifying the data type FIXED when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

FLOAT

Definition

FLOAT (p): A column with a floating point number [Page 16] with precision p (0<p<=38).

Use

Specifying the data type FLOAT when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

INT[EGER]

Definition

INT[EGER]: This data type is the same as FIXED [Page 121](10.0). Its permitted values are between -2147483648 and 2147483647.

Use

Specifying the data type INT[EGER] when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

SMALLINT

Definition

SMALLINT: This data type is the same as FIXED [Page 121](5.0). Its permitted values are between -32768 and 32767.

Use

Specifying the data type SMALLINT when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

DATE

Definition

DATE: An alphanumeric column in which you can store date values [Page 17].

Use

Specifying the data type DATE when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

TIME

Definition

TIME: An alphanumeric column in which you can store time values [Page 17].

Use

Specifying the data type TIME when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

TIMESTAMP

Definition

TIMESTAMP: An alphanumeric column in which you can store time stamp values [Page 17].

Use

Specifying the data type TIMESTAMP when defining columns [Page 118]

Integration

Data type (data_type) [Page 119]

Memory requirements of a column value per data types

Data type [Page 119]

Column definition [Page 118]

Data Type	Memory Requirements of a Column Value in Bytes for This Data Type
FIXED [Page 121](p,s)	(p+1) DIV 2 + 2
FLOAT [Page 122](p)	(p+1) DIV 2 + 2
BOOLEAN [Page 121]	2
DATE [Page 122]	9
TIME [Page 123]	9
TIMESTAMP [Page 123]	21
LONG [Page 121]	9
CHAR [Page 120](n); n<=30	n+1
CHAR(n); 30<n<=254; key column	n+1
CHAR(n); 30<n<=254; not key column	n+2
CHAR(n); 254<n	n+3
CHAR(n) UNICODE; n<=15	2*n+1
CHAR(n) UNICODE; 15<n<=127; key column	2*n+1
CHAR(n) UNICODE; 15<n<=127; not key column	2*n+2

CHAR(n) UNICODE; 127<n	2*n+3
VARCHAR [Page 120](n); 30<n<=254; key column	n+1
VARCHAR(n); 30<n<=254; not key column	n+2
VARCHAR(n); 254<n	n+3
VARCHAR(n) UNICODE; 15<n<=127; key column	2*n+1
VARCHAR(n) UNICODE; 15<n<=127; not key column;	2*n+2
VARCHAR(n) UNICODE; 127<n	2*n+3

Column Attributes (column_attributes)

A column_definition [Page 118] can contain the column name and column attributes.

Syntax

```
<column_attributes> ::= [<key_or_not_null_spec>] [<default_spec>]
[UNIQUE] [<constraint_definition>]
[REFERENCES <referenced_table> [(<referenced_column>)]
[<delete_rule>]]

<key_or_not_null_spec> ::= [PRIMARY] KEY | NOT NULL [WITH DEFAULT]
```

default_spec [Page 125], constraint_definition [Page 127], delete_rule [Page 130]

- A CONSTRAINT definition [Page 127] defines a condition that must be fulfilled by all the column values in the columns defined by the column definition [Page 118].
- `[REFERENCES <referenced_table> [(<referenced_column>)]`
  `[<delete_rule>]`
  has the same effect as specifying the referential CONSTRAINT definition [Page 128]
  `FOREIGN KEY [<referential_constraint_name>]`
  `(<referencing_column>)`
  `REFERENCES <referenced_table> [(<referenced_column>,...)]`
  `[<delete_rule>]`

`referenced_table` `referenced_column`	referenced table referenced column

Explanation

The [PRIMARY] KEY and UNIQUE column attributes must not be used together in a column definition.

If the [PRIMARY] KEY column attribute is specified, the CREATE TABLE statement [Page 115] must not contain a key definition [Page 131].

LONG data type: you may only specify NOT NULL or a DEFAULT specification [Page 125] as a column attribute for LONG [Page 121] columns.

UNIQUE

The UNIQUE column attribute determines the uniqueness of column values (see also CREATE INDEX statement [Page 148]).

KEY

If the KEY column attribute is specified, this column is part of the key of a table and is called the key column. The database system ensures that the key values in a table are unique. To

improve performance, the key should start with key columns which can assume many different values and which are to be used frequently in conditions with the "=" operator.

See also:

Restrictions [Page 219]

If a table is defined without a key column, the database system implicitly creates a key column SYSKEY CHAR(8) BYTE. This column is not visible with a SELECT *. However, it can be specified explicitly and has then the same function as a key column. The SYSKEY column can be used to obtain unique keys generated by the database system. The keys are in ascending order, thus reflecting the order of insertion in the table. The key values in the SYSKEY column are only unique within a table; i.e., the SYSKEY column in two different tables may contain the same values. If a unique key is desired across the entire database system, a key column of the data type CHAR(8) BYTE with the DEFAULT specification [Page 125] STAMP can be defined.

NOT NULL

NOT NULL must not be used together with the DEFAULT specification [Page 125] DEFAULT NULL.

NOT NULL WITH DEFAULT defines a default value that is dependent on the data type of the column. NOT NULL WITH DEFAULT must not be used with any of the DEFAULT specifications.

Column data type	DEFAULT value
CHAR [Page 120](n); VARCHAR [Page 120](n)	' '
CHAR(n) BYTE; VARCHAR(n) BYTE	X'00'
FIXED [Page 121](p,s), INT [Page 122], SMALLINT [Page 122], FLOAT [Page 122](p)	0
DATE [Page 122]	DATE
TIME [Page 123]	TIME
TIMESTAMP [Page 123]	TIMESTAMP
BOOLEAN [Page 121]	FALSE

DEFAULT Specification (default_spec)

A DEFAULT specification is formed by specifying the keyword DEFAULT and a DEFAULT value The maximum length of a default value is 254 characters.

Syntax

```
<default_spec> ::= DEFAULT <literal> | DEFAULT NULL
| DEFAULT USER | DEFAULT USERGROUP
| DEFAULT DATE | DEFAULT TIME | DEFAULT TIMESTAMP
| DEFAULT TRUE | DEFAULT FALSE
| DEFAULT TRANSACTION | DEFAULT STAMP
| DEFAULT SERIAL[(<unsigned_integer>)]
```

literal [Page 31], unsigned integer [Page 34]

Explanation

If a DEFAULT specification has been made for a column, the default value (<literal>, NULL, USER,...) must be a value that can be inserted in the column.

DEFAULT specification	Explanation
DEFAULT <literal>	The literal must be comparable with the data type of the column.
DEFAULT USER	Supplies the user name of the current user and can only be specified for columns of the data type [VAR]CHAR(n) (n>=32).
DEFAULT USERGROUP	Supplies only members of a usergroup, the usergroup name, or the user name for users that do not belong to a usergroup. This DEFAULT specification can only be specified for columns of the data type [VAR]CHAR(n) (n>=32).
DEFAULT DATE	Supplies the current date [Page 17] and can only be specified for columns of the data type DATE.
DEFAULT TIME	Supplies the current time [Page 17] and can only be specified for columns of the data type TIME.
DEFAULT TIMESTAMP	Supplies the current timestamp [Page 17] and can only be specified for columns of the data type TIMESTAMP.
DEFAULT TRUE/DEFAULT FALSE	Can only be specified for columns of the data type BOOLEAN [Page 17].
DEFAULT TRANSACTION	Supplies the identification of the current transaction [Page 210] and can only be specified for columns of the data type CHAR(n) BYTE (n>=8).
DEFAULT STAMP	Supplies a value of eight characters in length that is unique within the database system and can only be specified for columns of the data type CHAR(n) BYTE (n>=8). If a table is defined without a key column, the database system implicitly creates a key column SYSKEY CHAR(8) BYTE. The key values in the SYSKEY column are only unique within a table; i.e., the SYSKEY column in two different tables may contain the same values. If a unique key is desired across the entire database system, a key column can be defined with the DEFAULT specification STAMP.
DEFAULT SERIAL [(<unsigned_integer)]	Supplies a number generator for positive integers and can only be specified for columns of the data type INTEGER, SMALLINT, and FIXED without decimal places (SERIAL [Page 22]). The first value generated by the generator can be defined by specifying an unsigned integer (must be greater than 0). If this definition is missing, 1 is defined as the first value. If the value 0 is inserted in this column by an INSERT statement, the current number generator value is supplied and not the value 0. Each table may not contain more than one column with the DEFAULT specification DEFAULT SERIAL.

CONSTRAINT definition (constraint_definition)

A CONSTRAINT definition defines an integrity condition (restrictions for column values, see data integrity [Page 28]) that must be fulfilled by all the rows **in one** table.

Syntax

```
<constraint_definition> ::= CHECK <search_condition>
| CONSTRAINT <search_condition>
| CONSTRAINT <constraint_name> CHECK <search_condition>
```

search condition [Page 70], constraint name [Page 40]

Simple constraint (for one column), example table customer [Page 111]

```
title CHAR(7) CONSTRAINT title IN ('Mr','Mrs','Comp')
```

Complex constraint (for several columns), example table reservation [Page 114]

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

The system checks whether the arrival is before the departure.

Explanation

A CONSTRAINT definition defines an integrity condition that must be fulfilled by all the column values in the columns defined by the column definition [Page 118] with CONSTRAINT definition.

The CONSTRAINT definition in a column is checked when a row is inserted and a column changed that occurs in the CONSTRAINT definition. If the CONSTRAINT definition is violated, the INSERT or UPDATE statement fails.

When you define a constraint, you specify implicitly that the NULL value is not permitted as an input.

The search condition (`search_condition`) of the CONSTRAINT definition must not contain a subquery [Page 200].

The search condition of the CONSTRAINT definition must only contain column names in the form `<column name>`.

Constraint name

- No constraint name:
  The database system assigns a constraint name that is unique for the table in question.
- Constraint name is specified:
  The constraint name must be different to all other constraint names for this table.

Number of columns in a search condition

- Contains only one column name in the table:
  When the table is created (CREATE TABLE statement [Page 115]), you can check whether an additional DEFAULT value (default_spec [Page 125]) specified as a column attribute fulfills the search condition. If it is not true, the CREATE TABLE statement fails.
- Contains more than one column name in the table:
  When the table is created (CREATE TABLE statement), it is not possible to decide whether DEFAULT values of the table columns fulfill the search condition. In this case,

an attempt to insert DEFAULT values in the table when an INSERT or UPDATE statement is executed may fail.

Referential CONSTRAINT definition (referential_constraint_definition)

A referential CONSTRAINT definition defines an integrity condition (restrictions for columns values, see data integrity [Page 28]) that must be satisfied by all the rows in **two** tables. The resultant dependency between two tables affects changes to the rows contained in them.

Syntax

```
<referential_constraint_definition> ::=
FOREIGN KEY [<referential_constraint_name>]
(<referencing_column>,...)
REFERENCES <referenced_table> [(<referenced_column>,...)]
[<delete_rule>]
```

referential_constraint_name [Page 44], delete_rule [Page 130]

`referenced_table` `referenced_column`	Reference table, referenced column (table/column that is to be addressed)
`referencing_column`	Referencing column (column that establishes the link to the column that is to be addressed)

Dependency between the model tables customer [Page 111] and reservation [Page 114]. The referential CONSTRAINT definition is specified when the `reservation` table is defined. The `reservation` table is assigned a foreign key that corresponds to the key in the `customer` table.

```
CREATE TABLE reservation (rno FIXED(4) PRIMARY KEY, cno
FIXED(4), hno FIXED(4), roomtype CHAR(6), arrival DATE,
departure DATE,

FOREIGN KEY customer_reservation (cno) REFERENCES customer
ON DELETE CASCADE)
```

The defined relationship is called `customer_reservation`. The DELETE rule `ON DELETE CASCADE` specifies that deleting rows in the customer table causes the associated rows in the reservation table to be deleted automatically.

Explanation

A referential CONSTRAINT definition can be used in a CREATE TABLE statement [Page 115] or ALTER TABLE statement [Page 133]. The table specified in the corresponding statement (`table_name`) is referred to in the following sections as the referencing table.

The referencing columns are specified in the referential CONSTRAINT definition. The referencing columns must denote columns in the referencing table and must all be different. They are also called **foreign key columns**.

Referenced columns (`referenced_column`)

- If no referencing columns are specified, the result is the same as if the key columns in the referenced table were specified in the defined sequence.
- If referenced columns are specified that are not the key in the referencing table, the referenced table must have a UNIQUE definition [Page 132] whose column names and sequence match those of the referenced columns.

Relationship between referenced and referencing columns:

- The number of referenced columns is equal to the number of referencing columns.
- The nth referencing column corresponds to the nth referenced column.
- The data type and the length of each referencing column must match the data type and length of the corresponding referenced column.

The referencing table and the referenced table must be base tables, but not temporary base tables.

The current user must have the ALTER privilege for the referencing table and the REFERENCE privilege for the referenced table.

Name of a referential constraint (`referential_constraint_name`)

The name of a referential constraint [Page 44] can be specified after the keywords `FOREIGN KEY`.

- If the **name of a referential constraint is specified**, it must be different from all other names of referential constraints for the referencing table.
- If **no referential constraint name is specified**, the database system assigns a unique name (based on the referencing table).

Inserting and modifying rows in the referenced table

The following restrictions apply when rows in the referencing table are added or modified:

Let Z be an inserted or modified row. Rows can only be inserted or modified if one of the following conditions is fulfilled for the associated referenced table:

- Z is a matching row [Page 131]
- Z contains a NULL value in one of the referencing columns.
- The referential CONSTRAINT definition defines the DELETE rule ON DEFAULT SET DEFAULT, and Z contains the DEFAULT value in each referencing column.

Further terms

- DELETE rule [Page 130]
- CASCADE dependency [Page 130]
- Reference cycle [Page 131]
- A referential CONSTRAINT definition is **self-referencing** if the referenced and referencing tables are identical.
  With self-referencing referential CONSTRAINT definitions, the order in which a DELETE statement is processed can be important.
  Specifying CASCADE: all of the rows affected by the DELETE statement are first deleted irrespective of the referential CONSTRAINT conditions. All matching rows in the rows that have just been deleted are then also deleted. As a result, all of the matching rows in the previous deletion operation are deleted, etc.
  Specifying SET NULL or SET DEFAULT: all of the rows affected by the DELETE statement are first deleted irrespective of the referential CONSTRAINT conditions. Following this, SET NULL or SET DEFAULT is applied to the matching row.
- When rows are deleted from a referenced table, the number of rows deleted is entered in the third SQLERRD entry in the SQLCA database.
- When an INSERT or UPDATE statement is applied to a referencing table, irrespective of the isolation level defined for the current session, the database uses a blocking behavior for the referenced table that corresponds to isolation level 1.
  When a DELETE statement is applied to a referenced table, the database system uses a locking behavior that corresponds to isolation level 3.

DELETE rule

The DELETE rule defines the effects that deleting a row in the referenced table has on the referencing table (see referential_constraint_definition [Page 128]). The DELETE rule is also used when column attributes [Page 124] are defined.

Syntax

```
<delete_rule> ::= ON DELETE CASCADE | ON DELETE RESTRICT
| ON DELETE SET DEFAULT | ON DELETE SET NULL
```

Explanation

- No DELETE rule: deleting a row in the referenced table will fail if matching rows [Page 131] exist.
- ON DELETE CASCADE: if a row in the referenced table is deleted, all of the matching rows are deleted.
- ON DELETE RESTRICT: deleting a row in the referenced table will fails if matching rows exist.
- ON DELETE SET DEFAULT: if a row in the referenced table is deleted, the associated DEFAULT value is assigned to each referencing column for each matching row.
  A DEFAULT specification [Page 125] must exist for each referencing column.
- ON DELETE SET NULL: if a row in the referenced table is deleted, a NULL value is assigned to each referencing column of every matching row.
  None of these referencing tables may be a NOT NULL column.

CASCADE dependency

A table T* is CASCADE dependent on table T if a series of referential CONSTRAINT definitions [Page 128] R1, R2, ..., Rn (n>=1) exist where:

- T* is the referencing table of R1
- T is the referenced table of Rn
- All of the referential CONSTRAINT definitions use CASCADE from the DELETE rule [Page 130] or from the CASCADE option [Page 132].
- For i=1,...,n-1, n>1 is the referenced table of Ri is equal to the referencing table of Ri+1

Let R1 and R2 be two different referential CONSTRAINT definitions with the same referencing table S. T1 denotes the referenced table of R1, T2 denotes the referenced table of R2.

If T1 and T2 are identical, or if a table T exists so that T1 and T2 are CASCADE dependent on T, then R1 and R2 must both specify either CASCADE or RESTRICT.

There are different sequences of referential CONSTRAINT definitions that link the tables S and T. A DELETE statement on table T results in an action in table S. In order to ensure that the result of the DELETE statement does not depend on which of the two sequences of referential CONSTRAINT definitions is processed, the above restriction was selected for R1 and R2.

See also:

CASCADE option [Page 132]

Reference cycle

A reference cycle is a sequence of referential CONSTRAINT definitions [Page 128] R1, R2,...,Rn where n>1, so that the following applies:

- i=1,...,n-1 the referenced table of Ri is equal to the referencing table of Ri+1
- the reference table of Rn is the referencing table of R1

A reference cycle in which all of the referential CONSTRAINT definitions specify CASCADE (CASCADE dependency [Page 130]) is not allowed.

A reference cycle in which one referential CONSTRAINT definition does not specify CASCADE and all other referential CONSTRAINT definitions specify CASCADE is not allowed.

Matching row

A row in the referencing table is called a matching row of a row in the referenced table if the values of the corresponding referencing and referenced columns are identical.

A referential CONSTRAINT definition [Page 128] defines a 1:n relationship between two tables. This means that more than one matching row can exist for each row in the referenced table.

A row in the referenced table in a referenced column cannot be changed if at least one matching row exists.

Key Definition (key_definition)

A key definition in a CREATE TABLE statement [Page 115] or an ALTER TABLE statement [Page 133] defines the key in a base table. The key definition is introduced by the keywords `PRIMARY KEY`.

Syntax

```
<key_definition> :: PRIMARY KEY (<column_name>,...)
```

column_name [Page 46]

SQL statement for creating a `person` table with a one-column primary key for the column `cno`:

```
CREATE TABLE person (cno FIXED(4), firstname CHAR(7), name
CHAR(7), account FIXED(7,2), PRIMARY KEY (cno))
```

Rows are inserted in the same way as in a base table without a key definition. Double entries for the customer number, however, are rejected.

Explanation

The column name must identify a column in the base table. The specified column names are key columns in the table.

A key column must not identify a column of the data type LONG [Page 121] and is always a NOT NULL column. The database system ensures that no key column has a NULL value and that no two rows of the table have the same values in all key columns.

See also:

Restrictions [Page 219]

UNIQUE Definition (unique_definition)

A UNIQUE definition in the CREATE TABLE statement [Page 115] defines the uniqueness of column value combinations.

Syntax

```
<unique_definition> ::= [CONSTRAINT <index_name>] UNIQUE
(<column_name>,...)
```

index_name [Page 42], column_name [Page 46]

Explanation

Specifying a UNIQUE definition in a CREATE TABLE statement has the same effect as specifying the CREATE TABLE statement without a UNIQUE definition, but with a CREATE INDEX statement [Page 148] with UNIQUE.

- **Index name specified**: the generated index is stored under this name in the database catalog.
- **No index name**: The database system assigns a unique index name to the index.

DROP TABLE statement

A DROP TABLE statement deletes a base table (see Table [Page 23]).

Syntax

```
<drop_table_statement> ::= DROP TABLE <table_name> [<cascade_option>]
```

table_name [Page 47], cascade_option [Page 132]

Explanation

The table name must be the name of an existing base table. The current user must be the owner of the base table.

All the metadata and rows in the base table and all the view tables (see table [Page 23]), indexes [Page 24], privileges [Page 25], synonyms [Page 24], and referential CONSTRAINT definitions [Page 128] derived from it are deleted.

CASCADE option RESTRICT: the DROP TABLE statement will fail if view tables or synonyms are based on the specified table.

No CASCADE option specified: the CASCADE value is accepted.

If all of the data that is linked to this base table by means of a referential_constraint_definition [Page 128] with a DELETE rule [Page 130], are processed according to the specified DELETE rule, a DELETE statement [Page 181] must first be executed for this base table and then the DROP TABLE statement.

CASCADE option

A CASCADE option determines the deletion behavior for objects (e.g. tables, users), i.e. it defined whether certain dependencies are to be taken into account when objects are deleted.

Syntax

```
<cascade_option> ::= CASCADE | RESTRICT
```

See also:

CASCADE dependency [Page 130]

ALTER TABLE statement

An ALTER TABLE statement changes the properties of a base table (see Table [Page 23]).

Syntax

```
<alter_table_statement> ::=
  ALTER TABLE <table_name> <add_definition>
| ALTER TABLE <table_name> <drop_definition>
| ALTER TABLE <table_name> <alter_definition>
| ALTER TABLE <table_name> <column_change_definition>
| ALTER TABLE <table_name> <modify_definition>
| ALTER TABLE <table_name> <referential_constraint_definition>
| ALTER TABLE <table_name> DROP FOREIGN KEY
<referential_constraint_name>
| ALTER TABLE <table_name> <sample_definition>
```

table_name [Page 47], add_definition [Page 133], drop_definition [Page 136], alter_definition [Page 134], column_change_definition [Page 135], modify_definition [Page 137], referential_constraint_definition [Page 128], referential_constraint_name [Page 44], sample_definition [Page 117]

Explanation

The table name must be the name of an existing base table. The table must not be a temporary base table. The current user must have the ALTER privilege for the specified table.

- If a referential CONSTRAINT definition was specified, a new referential constraint is defined for the base table. The rules described in the referential CONSTRAINT definition [Page 128] apply.
- If DROP FOREIGN KEY was specified, the referential CONSTRAINT definition identified by the name of the referential constraint is dropped.
- If a SAMPLE definition is specified, a new number of rows is defined and is taken into account by the database system when the table statistics are calculated.

ADD Definition (add_definition)

You can define additional table properties by specifying an ADD definition in the ALTER TABLE statement [Page 133].

Syntax

```
<add_definition> ::= ADD <column_definition>,...
| ADD (<column_definition>,...)
| ADD <constraint_definition>
| ADD <referential_constraint_definition>
| ADD <key_definition>
```

column_definition [Page 118], constraint_definition [Page 127], key_definition [Page 131], referential_constraint_definition [Page 128]

The following statement adds two columns to the customer [Page 111] table. The columns initially contain the NULL value [Page 15] in all rows.

```
ALTER TABLE customer ADD (phone_number FIXED (8), street
CHAR (15))
```

The new columns can be used straight away.

Explanation

Adding a column definition: ADD <column_definition>

You can extend the table specified in the ALTER TABLE statement to include these columns by specifying column definitions [Page 118]. These specifications must not exceed the maximum number of columns allowed and the maximum length of a row.

The memory requirements for each column are increased by one character (for normal memory requirements, see Memory requirements of a column value as a function of the data type [Page 123]), if the length described is less than 31 characters and the column does not have the data type VARCHAR [Page 120].

The newly defined columns contain the NULL value in all rows, if no default value has been specified for these columns using a DEFAULT specification [Page 125]. If the NULL value violates a CONSTRAINT definition [Page 127] of the table, the ALTER TABLE statement will fail.

When you add LONG columns, you cannot make a DEFAULT specification. If you want to define a default value for a LONG column, you can define the LONG column using the ADD definition and then use MODIFY Definition [Page 137] to define a default value for the definition.

In every other respect, specifying a column definition has the same effect as specifying a column definition in a CREATE TABLE statement [Page 115].

- If view tables are defined for the specified table, and if alias names [Page 39] are defined for one of these view tables, and if the view tables reference the columns in the table with *, the ALTER TABLE statement will fail.
- If view tables are defined for the specified table, and if no alias names are defined, and if the view tables reference the columns in the table with *, this view table contains the columns added to the base table by the ADD definition.

Adding a CONSTANT definition: ADD <constant definition>

All of the rows in the table must satisfy the condition defined by the search condition [Page 70] of the CONSTRAINT definition [Page 127].

Adding a referential CONSTRAINT definition: ADD <referential_constraint_definition>

An integrity condition is defined for the table specified in the ALTER TABLE statement. The columns specified in the referential CONSTRAINT definition [Page 128] must be columns in the table. All of the rows in the table must satisfy the integrity condition defined by the referential CONSTRAINT definition.

Adding a key definition: ADD <key definition>

A key is defined for the table specified in the ALTER TABLE statement. At execution time, the table must only contain the key column SYSKEY generated by the database system. The columns specified in the key definition [Page 131] must be columns in the table and must satisfy the key properties (none of the columns may contain NULL value, and no two rows in the table may have the same values in all columns of the key definition). The new key is stored in the metadata of the table. The key column SYSKEY is omitted. This is an extremely lengthy procedure for tables with a large number of rows, since extensive copy operations are carried out.

ALTER definition

By specifying an ALTER definition in the ALTER TABLE statement [Page 133], you can change a CONSTRAINT definition [Page 127] or a key definition [Page 131] .

Syntax

```
<alter_definition> ::=
  ALTER CONSTRAINT <constraint_name> CHECK <search_condition>
| ALTER <key_definition>
```

constraint_name [Page 40], search_condition [Page 70], key_definition [Page 131]

Explanation

CONSTRAINT <constraint_name>

The constraint name must identify a CONSTRAINT definition in the table. If the specified search condition is not violated by any row in the table, it replaces the existing search condition of the CONSTRAINT definition. Otherwise, the ALTER TABLE statement fails.

<key_definition>

The key specified by the key definition replaces the current key in the table. The columns specified in the key definition must identify columns in the table and must have the key property (key definition [Page 131]).

If a column of the key to be replaced is a referenced column of a referential CONSTRAINT definition, the ALTER TABLE statement will fail.

With large tables, in particular, this may take more time, since extensive copy operations have to be carried out.

COLUMN change definition

You can modify the properties of a column by specifying a COLUMN change definition in the ALTER TABLE statement [Page 133].

Syntax

```
<column_change_definition> ::= COLUMN <column_name> NOT NULL
| COLUMN <column_name> DEFAULT NULL
| COLUMN <column_name> ADD <default_spec>
| COLUMN <column_name> ALTER <default_spec>
| COLUMN <column_name> DROP DEFAULT
```

column_name [Page 46], default_spec [Page 125]

Explanation

NOT NULL

`NOT NULL` can only be specified if the column contains no NULL value [Page 15]s. You cannot add a NULL value to the column once the ALTER TABLE statement has been successfully executed.

DEFAULT NULL

`DEFAULT NULL` allows a NULL value for the column. The system does not check whether a NULL value violates existing CONSTRAINT definitions [Page 127] in the table. For this reason, inserting the NULL value can fail when an INSERT or UPDATE statement is executed.

ADD <default_spec>

The column must not contain a DEFAULT specification before the ALTER TABLE statement is executed with `ADD <default spec>`. `ADD <default spec>` assigns a DEFAULT value to the column.

ALTER <default spec>

`ALTER <default spec>` changes the DEFAULT value assigned to the column. All of the rows that contain the old default value in the column remain unaltered.

DROP DEFAULT

`DROP DEFAULT` drops the DEFAULT specification of the column. If the column is the foreign key column of a referential CONSTRAINT definition [Page 128] with the DELETE RULE [Page 130] ON DELETE SET DEFAULT, the ALTER TABLE statement will fail.

DROP definition

You can delete table properties by specifying a DROP definition in the ALTER TABLE statement [Page 133].

Syntax

```
<drop_definition> ::= DROP <column_name>,... [<cascade_option>]
[RELEASE SPACE]
| DROP (<column_name>,...) [<cascade_option>] [RELEASE SPACE]
| DROP CONSTRAINT <constraint_name> | DROP PRIMARY KEY
```

column_name [Page 46], cascade_option [Page 132], constraint_name [Page 40]

Explanation

Dropping a column: DROP <column_name>

Each column name must be a column of the table identified by the ALTER TABLE statement. The column must be neither a key column nor a foreign key column of a referential CONSTRAINT definition [Page 128] of the table.

The columns are marked as dropped in the metadata of the table. A DROP definition does not automatically reduce the memory requirements of the underlying table. `RELEASE SPACE` forces the column values of the dropped columns to be dropped in every row in the table. With large tables, in particular, this may take more time, since extensive copy operations have to be carried out.

Any privileges and comments for the columns to be dropped are dropped as well.

If one of the columns to be dropped occurs as a selected column [Page 193] in a view definition, the specified column in the view table is dropped.
If this view table is used in the FROM condition of another view table, the described procedure is recursively applied to this view table.

- If one of the columns to be dropped occurs in the QUERY specification [Page 192] of a view definition and if no CASCADE condition [Page 132] is specified or if the CASCADE condition in the DROP is specified in the DROP definition, the view definition is dropped with all the view tables, privileges, and synonyms that depend on it.
- The ALTER TABLE statment will fail if one of the columns to be dropped appears in the QUERY specification of a view definition and the CASCADE condition RESTRICT is specified in the DROP definition.

Existing indexes referring to columns to be dropped are also dropped. The storage locations for the dropped indexes are released.

All CONSTRAINT definitions [Page 127] that contain one of the dropped columns are dropped.

Dropping a constraint: DROP CONSTRAINT <constraint_name>

The constraint name must identify a CONSTRAINT definition in the table. The latter is then removed from the metadata of the table.

Dropping a key: DROP PRIMARY KEY

- The table must have a key defined by the user.
- The table must not contain more than 1023 columns.
- The maximum permissible length of a row must not exceed 8088 bytes.
- The key columns must not be a referenced column of a referential CONSTRAINT definition [Page 128].

The key is replaced by the key column SYSKEY generated by the database system. With large tables, in particular, this may take more time, since extensive copy operations have to be carried out.

MODIFY definition

You can modify data types and properties of table columns by specifying a MODIFY definition in the ALTER TABLE statement [Page 133].

Syntax

```
<modify_definition> ::= MODIFY (<column_name> [<data_type>]
[<column_attributes>]...)
```

column_name [Page 46], data_type [Page 119], column_attributes [Page 124]

The parentheses are not necessary if the MODIFY definition only contains one column name.

Explanation

Each column name must be a column of the base table specified in the ALTER TABLE statement.

Column attributes

Only the following column attributes are allowed:

- `NULL`
  NULL value [Page 15]
- `NOT NULL`
  If NOT NULL is specified, the table must not have any rows that contain a NULL value in the corresponding column. A NULL value can no longer be inserted into the column after being modified.
- DEFAULT specification [Page 125]
  The DEFAULT specification DEFAULT SERIAL is not permitted.
  If a DEFAULT specification is specified, it replaces an existing DEFAULT specification in the corresponding column. The new DEFAULT specification only affects subsequent INSERT statements and not affect rows that already exist in the table.

Data types

If a DEFAULT specification is not specified and if a DEFAULT specification is defined for the corresponding column, it must be compatible with the data type.

- **Code attribute ASCII or EBCDIC**: the corresponding column must have the data type DATE [Page 122], TIME [Page 123], or TIMESTAMP [Page 123] or the code attribute [Page 17] ASCII, EBCDIC, or UNICODE before it is modified.

- **Code attribute UNICODE**: A transformation from UNICODE to ASCII must be possible for the relevant column.
- **Code attribute BYTE**: the corresponding column must have the data type DATE, TIME, or TIMESTAMP or the code attribute ASCII, EBCDIC, or BYTE before it is modified.

Data type **CHAR(n)**, **VARCHAR(n)**: the corresponding column must have the data type CHAR [Page 120](n), VARCHAR [Page 120](n), DATE, TIME, or TIMESTAMP. In this case, the table must not contain a row in which the column has a value with a length greater than n. If a column had the code attribute UNICODE before the ALTER TABLE statement was executed, and the new code attribute is not UNICODE, transformation to the new code attribute must be possible.

Data type **DATE**: the corresponding column must have the data type CHAR(n), VARCHAR(n), or DATE [Page 122]. This column must contain a date value in any of the date formats supported by the database system in all rows of the table.

Data type **FIXED(n,m)**: the corresponding column must have the data type FIXED [Page 121](n,m), FLOAT [Page 122], INT [Page 122], or SMALLINT [Page 122]. In this case, the table must not contain a row in which the column has a value with more than (n - m) integral or m fractional digits.

Data type **FLOAT(n)**: the corresponding column must have the data type FIXED(n,m), FLOAT(n), INT, or SMALLINT.

Data type **INT**: the corresponding column must have the data type FIXED(n,m), FLOAT(n), INT, or SMALLINT. In this case, the table must only contain rows in which this column has integral values in the range between -2147483648 and 2147483647.

Data type **SMALLINT**: the corresponding column must have the data type FIXED(n,m), FLOAT(n), INT, or SMALLINT. In this case, the table must only contain rows in which this column has integral values in the range between -32768 and 32767.

Data type **TIME**: the corresponding column must have the data type CHAR(n), VARCHAR(n), or TIME [Page 123]. This column must contain a time value in any of the time formats supported by the database system in all rows of the table.

Data type **TIMESTAMP**: the corresponding column must have the data type CHAR(n), VARCHAR(n), or TIMESTAMP [Page 123]. This column must contain a timestamp value in any of the timestamp formats supported by the database system in all rows of the table.

Column attribute **NULL**: a NULL value [Page 15] can be entered in the corresponding column with a subsequent INSERT or UPDATE statement.

If one of the columns identified by `column name` is contained in a search condition [Page 70] for the table, this column must also define a legal search condition after the data type has been modified.

Others

Depending on the type of modification, the MODIFY definition may result in the table having to be recopied and/or indexes rebuilt. In such a case, the runtime will be considerably long.

If a table is recopied and the table contains columns marked as deleted, then these columns are removed from the catalog and from the table rows, thus reducing the space requirements of the table.

RENAME TABLE statement

A RENAME TABLE statement changes the name of a base table (see Table [Page 23]).

Syntax

```
<rename_table_statement> ::=
RENAME TABLE <old_table_name> TO <new_table_name>

<old_table_name> ::= <table_name>
<new_table_name> ::= <identifier>
```

table_name [Page 47], identifier [Page 36]

Explanation

The old table name must identify a base table that is not a temporary table. The current user must be the owner of the table.

The new table name must not already be assigned to a base or view table or a private synonym [Page 24] of the current user.

The old table is assigned the name specified in the `new_table_name`. All of the properties of the table (e.g. privileges, indexes) remain unchanged. The definitions of view tables based on the old table name are adapted to the new name.

RENAME COLUMN statement

A RENAME COLUMN statement (`rename_column_statement`) changes the name of a table column.

Syntax

```
<rename_column_statement> ::=
RENAME COLUMN <table_name>.<column_name> TO <column_name>
```

table_name [Page 47], column_name [Page 46]

Explanation

The specified table [Page 23] must be a base or view table. The current user must be the owner of the table.

The specified table column is given a new name. If the column name of a view table (that was defined with this table) was derived from the column name of the base table, the old column name in the view table is replaced by the new name. If the new column name is identical with an existing column name of the view table, the RENAME COLUMN statement fails.

EXISTS TABLE statement

An EXISTS TABLE statement indicates whether a table exists or not.

Syntax

```
<exists_table_statement> ::= EXISTS TABLE <table_name>
```

table_name [Page 47]

Explanation

The specified table [Page 23] must be a base table, view table, or a synonym [Page 24].

The existence or non-existence of the specified table is indicated by the return code 0 or by the error message `-4004 UNKNOWN TABLE NAME`.

A table only exists for a user if the user has a privilege on this table.

CREATE DOMAIN statement

A CREATE DOMAIN statement defines a value range (domain) [Page 24].

Syntax

```
<create_domain_statement> ::= CREATE DOMAIN <domain_name> <data_type>
[<default_spec>] [<constraint_definition>]
```

domain_name [Page 41], data_type [Page 119], default_spec [Page 125], constraint_definition [Page 127]

Explanation

The CONSTRAINT definition must not contain a constraint name [Page 40].

The CREATE DOMAIN statement can be executed by all users with DBA status.

If the domain name is specified without an owner, the current user is assumed to be the owner. If you specify the domain name with an owner, this owner must be identical to the current user. In this case, the current user becomes the owner of the domain.

The name of the domain must differ from all other domain names of the current user.

If a domain is generated with a CONSTRAINT definition, the domain name is included in the search condition [Page 70] as a column name.

DROP DOMAIN statement

A DROP DOMAIN statement drops the definition of a domain [Page 24].

Syntax

```
<drop_domain_statement> ::= DROP DOMAIN <domain_name>
```

domain_name [Page 41]

Explanation

The domain name must identify an existing domain. The current user must be owner of the domain.

The metadata of the domain is dropped from the catalog. Dropping a domain has no effect on tables in which this domain was used to define columns.

CREATE SEQUENCE Statement (create_sequence_statement)

The CREATE SEQUENCE statement defines a database object that supplies integer values (number generator). In the following description, this object is referred to as a sequence.

Syntax

```
<create_sequence_statement> ::= CREATE SEQUENCE
[<owner>.]<sequence_name>
[INCREMENT BY <integer>] [START WITH <integer>]
[MAXVALUE <integer> | NOMAXVALUE] [MINVALUE <integer> | NOMINVALUE]
[CYCLE | NOCYCLE]
[CACHE <unsigned_integer> | NOCACHE]
[ORDER | NOORDER]
```

owner [Page 41], sequence_name [Page 45], integer [Page 34], unsigned_integer [Page 34]

Explanation

The sequence names can be specified in any order.

The current user must have RESOURCE or DBA status. If an owner is specified, it must identify the current user. The current user becomes the owner of the sequence.

The integer values generated by the sequence can be used to assign key values.

INCREMENT BY

Defines the difference between the next sequence value and the last value assigned. A negative value for INCREMENT BY generates a descending sequence. The value 1 is used if no value is assigned.

START WITH

Defines the first sequence value. If no value is specified, the value specified for MAXVALUE or –1 is used for descending sequences and the value specified for MINVALUE or 1 for ascending sequences.

MINVALUE

Defines the smallest value generated by the sequence. If no value is defined for MINVALUE, the smallest integer value that can be represented with 38 digits is used.

MAXVALUE

Defines the largest value generated by the sequence. If no value is defined for MAXVALUE, the largest integer value that can be represented with 38 digits is used.

CYCLE/NOCYCLE

CYCLE: MINVALUE is produced for ascending sequences after MAXVALUE has been assigned. MAXVALUE is produced for ascending sequences after MINVALUE has been assigned.

NOCYCLE: a request for a sequence value fails if the end of the sequence has already been reached, i.e. if MAXVALUE has been assigned for ascending sequences or MINVALUE for descending sequences.

If neither CYCLE nor NOCYCLE is specified, NOCYCLE is assumed.

CACHE/NOCACHE

CACHE: access to the sequence can be accelerated because the defined number of sequence values is held in the memory.

NOCACHE: no sequence values are defined beforehand.

If neither CACHE nor NOCACHE is specified, CACHE 20 is assumed.

ORDER/NOORDER

Specifying ORDER or NOORDER has no effect.

Sequence values can be specified using CURRVAL and NEXTVAL (see value_spec [Page 49]). In this way, you can interrogate or increase the current counter value.

DROP SEQUENCE statement

A DROP SEQUENCE statement drops a number generator (sequence).

Syntax

```
<drop_sequence_statement> ::= DROP SEQUENCE [<owner>.]<sequence_name>
```

owner [Page 41], sequence_name [Page 45]

Explanation

The current user must be the owner. The sequence name must identify a sequence of the current user.

The metadata of the sequence is removed from the catalog.

CREATE SYNONYM statement

The CREATE SYNONYM statement defines a synonym [Page 24] (alternative name) of a table name [Page 47].

Syntax

```
<create_synonym_statement> ::= CREATE [PUBLIC] SYNONYM
[<owner>.]<synonym_name> FOR <table_name>
```

owner [Page 41], synonym_name [Page 46], table_name [Page 47]

Explanation

The table name must not denote a temporary base table (see Table [Page 23]). The user must have a privilege for the specified table. The current user must be the owner.

The synonym name can be specified anywhere instead of the table name. This has the same effect as specifying the table name for which the synonym was defined.

PUBLIC

If PUBLIC is specified, the synonym name must not be identical to the name of a synonym defined with PUBLIC. A synonym is generated that can be accessed by all users.

If PUBLIC is not specified, a private synonym is generated that is only known by the current user. In this case, the synonym name must not be identical to the name of an existing base table, view table, or a private synonym of the current user. If a synonym with the same name and the PUBLIC attribute exists, it cannot be accessed by the current user until the private synonym has been dropped.

DROP SYNONYM statement

The DROP SYNONYM statement drops a synonym [Page 24] (alternative name) of a table name [Page 47].

Syntax

```
<drop_synonym_statement> ::= DROP [PUBLIC] SYNONYM
[<owner>.]<synonym_name>
```

owner [Page 41], synonym_name [Page 46]

Explanation

The specified synonym name must identify an existing synonym of the current user.

If PUBLIC is specified, the synonym identified by the synonym name must be defined as PUBLIC.

The synonym definition is removed from the set of table name synonyms available to the user.

RENAME SYNONYM statement

The RENAME SYNONYM statement changes the name of a synonym [Page 24].

Syntax

```
<rename_synonym_statement> ::= RENAME [PUBLIC] SYNONYM
<old_synonym_name> TO <new_synonym_name>
```

old/new_synonym_name [Page 46]

Explanation

The old synonym name must have been generated by the current user.

If PUBLIC is specified, the old must be defined as PUBLIC.

A table of the current user with the new synonym name must not exist already.

CREATE VIEW Statement (create_view_statement)

The CREATE VIEW statement defines a view table (see Table [Page 23]). A view table never actually exists physically. Instead, it is formed from the rows of the underlying base table(s) when this view table is specified in an SQL statement.

Syntax

```
<create_view_statement> ::= CREATE [OR REPLACE] VIEW <table_name>
[(<alias_name>,...)] AS <query_expression> [WITH CHECK OPTION]
```

table_name [Page 47], alias_name [Page 39], query_expression [Page 189]

Explanation

When the CREATE VIEW statement is executed, metadata that describes the view table is stored in the catalog.

The view table is always identical to the table that would be obtained as the result of the QUERY expression. The QUERY expression must not contain a parameter specification [Page 48]. The QUERY expression must not reference a temporary table or a result table name [Page 41].

The table expressions [Page 195] of the QUERY specification [Page 192] in the QUERY statement of the CREATE VIEW statement must not contain a QUERY expression.

If a column selected [Page 193] by the QUERY statement is of the data type LONG, the FROM clause [Page 196] must contain exactly one table name that is based on exactly one base table.

The user must have the SELECT privilege for all columns occurring in the view definition. The user is the owner of the view table and has at least the SELECT privilege for it. The user may grant the SELECT privilege for any columns in the view table derived from columns for which the user is authorized to grant the SELECT privilege to others. The user has the INSERT, UPDATE, or DELETE privilege when he has the corresponding privileges for the tables on which the view table is based, and when the view table is updateable. The user may only

grant these privileges to others if he or she is authorized to grant the corresponding privilege for all tables on which the view table is based.

OR REPLACE

If OR REPLACE is not specified, the table name must not be identical to the name of an existing view table.

If OR REPLACE is specified, the table name may be identical to the name of an existing view table. In this case, the definition of the existing view table is replaced by the new definition. The database system then attempts to adapt privileges granted for the existing view table to the new view definition, with the result that the privileges for the view table usually remain unchanged. Privileges are only removed implicitly if conflicts occur that cannot be resolved by the database system. If there are major discrepancies between the two view definitions, the CREATE VIEW statement may fail in the following case: the CREATE VIEW statement of a view table based on the existing view table cannot be executed correctly for the new view definition.

Alias names (alias_name)

The column names of the view table must be unique. Otherwise, alias names must be specified for the result table generated by the QUERY expression. The number of alias names must be equal to the number of columns in the result table generated by the QUERY expression. If no alias names are specified, the column names of the result table generated by the QUERY expression are applied to the view table. The column descriptions for the view table are taken from the corresponding columns in the QUERY expression. The FROM clause of the QUERY expression can contain one or more tables.

WITH CHECK OPTION

If the CREATE VIEW statement contains a WITH CHECK OPTION, the owner of the view table must have the INSERT, UPDATE, or DELETE privilege for the view table.

Specifying WITH CHECK OPTION has the effect that the INSERT statement or UPDATE statement issued on the view table does not create any rows that could not be selected subsequently via the view table; i.e. the search condition [Page 70] of the view table must be true for any resulting rows.

The CHECK OPTION is inherited; i.e. if a view table V was defined WITH CHECK OPTION and V occurs in the FROM clause of an updateable view table V1, only those rows that can be selected using V can be inserted or altered using V1.

Further terms and information

- Complex View Table [Page 144]
- Updateable View Table [Page 145]
- INSERT Privilege for Owners of the View Table [Page 145]
- UPDATE Privilege for Owner of the View Table [Page 146]
- DELETE Privilege for Owner of the View Table [Page 146]
- Updateable Join View Table [Page 146]

Complex view table

A view table (see CREATE VIEW statement [Page 143]) is a complex view table if it satisfies one of the following conditions:

- The definition of the view table contains DISTINCT or GROUP BY or HAVING.
- The CREATE VIEW statement contains EXCEPT, INTERSECT, or UNION.

- The search condition [Page 70] in the QUERY expression [Page 189] of the CREATE VIEW statement contains a subquery [Page 200].
- The CREATE VIEW statement contains an outer join, that is an OUTER JOIN indicator in a JOIN predicate [Page 62] of the search condition.

Updateable View Table

A view table (see CREATE VIEW statement [Page 143]) is called updateable if it is not a complex view table [Page 144], and if it is not based on a complex view table.

For join view tables, that is, view tables whose FROM clause [Page 196] contains more than one table or join table, the following additional conditions must be satisfied:

- Each base table on which the view table is based has a key defined by the user.
- Referential CONSTRAINT definitions [Page 128] must exist between the base tables on which the view table is based.
- One of the base tables, on which the view table is based, is not a referenced table of a referential CONSTRAINT definition for a different base table of the view table. This table is the **key table** of the view table.
- For each base table on which the view table is based, there is a sequence of referential CONSTRAINT definitions so that the respective base table can be accessed from the key table.
- The referential CONSTRAINT definitions must be reflected as a JOIN predicate [Page 62] in the search condition [Page 70] of the CREATE VIEW statement, that is, the condition "key column = foreign key column" must exist for every column in each referential CONSTRAINT definition.
- The CREATE VIEW statement must contain either the primary key or foreign key column from each referential CONSTRAINT definition as the selected column [Page 193], but not both.
- The view table must be defined with WITH CHECK OPTION.

INSERT privilege for the owner of the view table

The owner [Page 41] of the view table (see CREATE VIEW statement [Page 143]) has the INSERT privilege, i.e. he or she can specify the view table as a table in which insertion is to be made in the INSERT statement if the following conditions are satisfied:

- The view table is updateable (updateable view table [Page 145]).
- The owner of the view table has the INSERT privilege for all tables in the FROM clause [Page 196] of the CREATE VIEW statement.
- The selected columns [Page 193] of the CREATE VIEW statement consist of table columns or column names, not expressions [Page 52] with more than one column name.
- The CREATE VIEW statement contains every mandatory column from all tables of the FROM clause as the selected column.

UPDATE privilege for the owner of the view table

The owner [Page 41] of the view table (see CREATE VIEW statement [Page 143]) has the UPDATE privilege for a column in the view table, i.e. he or she can specify the column as a the column to be updated in an UPDATE statement if the following conditions are satisfied:

- The view table is updateable (updateable view table [Page 145]).
- The owner of the view table has the UPDATE privilege for the table columns or the column name that defines the column.
- The column is defined by specifying table columns or by means of a column name, but not by an expression [Page 52] with more than one column name.

DELETE privilege for the owner of the view table

The owner [Page 41] of the view table (see CREATE VIEW statement [Page 143]) has the DELETE privilege for the view table, i.e. he or she can specify the view table as a table in which entries are to be deleted in the DELETE statement if the following conditions are satisfied:

- The view table is updateable (updateable view table [Page 145]).
- The owner of the view table has the DELETE privilege for all tables in the FROM clause [Page 196] of the CREATE VIEW statement.

Updateable join view table

It is assumed that the definition of the join table V (see CREATE VIEW statement [Page 143]) in the FROM clause [Page 196] contains the base tables T1,...,Tn (n>1).

- Let Ti and Tj be two base tables selected by V. Let Rij be a referential CONSTRAINT definition [Page 128] of Ti and Tj in which Ti is the referencing table and Tj the referenced table.
  Let PKj1,...,PKjm be the key columns of Tj.
  Let Fki1,...,Fkim be the corresponding foreign key columns of Ti.
  The referential CONSTRAINT definition is **relevant** for V if the JOIN predicate [Page 62] (PKj1=Fki1 AND ... AND PKjm=FKjm) is part of the search condition [Page 70] of V.
- Let Ti and Tj be two base tables selected by V and Rij a referential CONSTRAINT definition of Ti and Tj that is relevant for V.
  Ti is the **predecessor** of Tj (Ti<Tj) if Rij is the only referential CONSTRAINT definition of Ti and Tj that is relevant for V.
- Let Rij be a referential CONSTRAINT definition that is relevant for V.
  Rij defines a **1 : 1 relationship** between Ti and Tj if the foreign key columns of Rij make up the key columns of Tj.
- Let Rij be a referential CONSTRAINT definition that is relevant to V and s a key column of Tj or a foreign key column of this referential CONSTRAINT definition of Ti. The **column s can be derived from V** if exactly one of the following conditions is satisfied:
  - s is a selected column [Page 193] of V.
  – a key column or a foreign key column s' of a referential CONSTRAINT definition that is relevant to V exists that can be derived from V and the JOIN predicate s=s' is part of the search condition of V.
- A column v of V **corresponds** to a column s of a base table T if one of the following conditions is satisfied
  - v is the i^{th} column of V and s is the i^{th} selected column of V
  - v corresponds to a key column PK of Tj of a referential CONSTRAINT definition Rij that is relevant to V and s is the foreign key column of Ti that is assigned to PK

- v corresponds to a foreign key column FK of Ti of a referential CONSTRAINT definition Rij that is relevant to V and s is the key column of Tj that is assigned to FK.

A **join view table V is updateable** if the following conditions are satisfied:

- Each base table Ti (1 <= i <= n) has a key defined by the user.
- The database system must be able to determine a processing sequence for the underlying base tables; i.e. an order $T_{i1},\ldots,Tin$ of the tables T1,…,Tn must exist so that j < k can be deduced from Tij<Tik. The columns of V from which the key columns of Ti1 can be derived make up the key of V. Ti1 is called the key table of V. The order of the tables does not have to be unique.
- Starting with a row in the key table of V, it must be possible to assign each underlying base table exactly one row; that is, there is a sequence of tables Ti1,...,Tij for each table Tij (1 <= j <= n) such that Ti1 < .. < Tij
  This sequence is unique for each base table referred to by V.
- It must be possible to derive the key columns and foreign key columns of all referential CONSTRAINT definitions relevant to V from the columns of V.
- The join predicates needed to recognize the relevance of a referential CONSTRAINT definition must be specified in parts of the search condition defined with the WITH CHECK OPTION. If the view definition only contains base tables, this means that the view table must be defined WITH CHECK OPTION. If a view table V is derived from a view table V' and if V' was defined WITH CHECK OPTION, then V inherits the CHECK OPTION for the part of the qualification passed on by V'.

DROP VIEW statement

The DROP VIEW statement drops a view table (see Table [Page 23]).

Syntax

```
<drop_view_statement> ::= DROP VIEW <table_name> [<cascade_option>]
```

table name [Page 47], cascade option [Page 132]

Explanation

The table name must identify an existing view table.

The user must be the owner of the specified view table.

The metadata of the view table and all dependent synonyms [Page 24], view tables, and privilege [Page 25]s are dropped. The tables on which the view table was created remain unaffected.

If the CASCADE option RESTRICT is specified and other view tables or synonyms based on this view table exist, the DROP VIEW statement will fail.

RENAME VIEW statement

A RENAME VIEW statement changes the name of a view table (see Table [Page 23]).

Syntax

```
<rename_view_statement> ::= RENAME VIEW <old_table_name> TO
<new_table_name>
```

```
<old_table_name> ::= <table_name>
<new_table_name> ::= <table_name>
```
table_name [Page 47]

Explanation

The `old_table_name` must be a view table. The current user must be the owner of the view table.

The `new_table_name` must not yet be used for a table of the current user.

The CREATE VIEW statement [Page 143] of the `old_table_name` view table is adapted to the new name. The result of this adaptation can be retrieved from the table DOMAIN.VIEWDEFS.

The definitions of view tables based on the `old_table_name` are adapted to the new name.

CREATE INDEX Statement (create_index_statement)

The CREATE INDEX statement creates an index [Page 24] of a base table (see table [Page 23]).

Syntax

```
<create_index_statement> ::=
CREATE [UNIQUE] INDEX <index_name> ON <table_name> (<column_name>
[ASC|DESC],...)
```

table_name [Page 47], column_name [Page 46], index_name [Page 42]

Explanation

Indexes provide access to the table data using non-key columns. Maintaining these indexes, however, can be quite complex in the case of an INSERT, UPDATE, or DELETE statement.

The index is created across the specified table columns. The secondary key consists of the specified columns of the table, in the specified order.

- The specified table must be an existing base table, and not a temporary table. The index name must not be identical with an existing index name of the table.
- The column defined by the `column name` must be a column in the specified table. This column must not be a LONG column [Page 16]. All of the column name pairs must be different.
- The current user must have the INDEX privilege type [Page 43] for the columns.
- Restrictions [Page 219]

UNIQUE

If UNIQUE is specified, the database system ensures that no two rows of the specified table have the same values in the indexed columns. In this way, if two rows both contain the NULL value [Page 15] for all columns of an index, the two index values are not considered to be identical. If there is not at least one column that does not contain the NULL value, two rows that have the same value in all non-NULL columns are considered to be identical.

ASC | DESC

The index values are stored in ascending or descending order. If the specification of ASC or DESC is omitted, ASC is implicitly assumed.

DROP INDEX Statement (drop_index_statement)

The DROP INDEX statement drops an index [Page 24] for a base table (see table [Page 23]). The metadata of the index is dropped from the database catalog. The storage space occupied by the index is released.

Syntax

```
<drop_index_statement> ::= DROP INDEX <index_name> [ON <table_name>]
```

index_name [Page 42], table_name [Page 47]

Explanation

The specified index must exist and the specified table name must be the name of an existing base table.

`ON <table name>` is not necessary if the index name identifies an index unambiguously.

The current user must be the owner of the specified table or have the INDEX privilege for it.

ALTER INDEX Statement (alter_index_statement)

The ALTER INDEX (`alter_index_statement`) statement determines how an index [Page 24] is used in data queries.

Syntax

```
<alter_index_statement> ::= ALTER INDEX <index_name> [ON
<table_name>] ENABLE
| ALTER INDEX <index_name> [ON <table_name>] DISABLE
| ALTER INDEX <index_name> [ON <table_name>] INIT USAGE
```

index_name [Page 42], table_name [Page 47]

Explanation

When a CREATE INDEX statement [Page 148] is executed, an index is generated across the specified columns. This index is modified accordingly for all of the following SQL statements for data manipulation (INSERT statement [Page 173], UPDATE statement [Page 178], DELETE statement [Page 181]). With all other SQL statements in which individual rows in a table are specified, the database system can use this index to speed up the search for these rows.

ALTER INDEX ... DISABLE

The index can no longer be used for this search, however it continues to be changed by the use of the SQL statements INSERT, UPDATE, or DELETE.

ALTER INDEX ... ENABLE

The index can be used for the search again.

ALTER INDEX ... INIT USAGE

The column INDEX_USED in the system table DOMAIN.INDEXES is initialized with 0; that is, the count of how often an index is used is restarted.

RENAME INDEX statement

The RENAME INDEX statement changes the name of an index [Page 24].

Syntax

```
<rename_index_statement> ::= RENAME INDEX <old_index_name> [ON
<table_name>] TO <new_index_name>

<old_index_name> ::= <index_name>
<new_index_name> ::= <index_name>
```

table_name [Page 47], index_name [Page 42]

Explanation

The specified table name must be the name of an existing base table (see table [Page 23]).

The index identified by the `old_index_name` must exist. `ON <table_name>` is not necessary if this index is unique.

The current user must be the owner of the specified table or have the INDEX privilege for it.

The new index name must not be identical to an existing index name for the table.

COMMENT ON Statement (comment_on_statement)

The COMMENT ON statement (`comment_on_statement`) creates, alters, or drops a comment for a database object stored in the database catalog.

Syntax

```
<comment_on_statement> ::= COMMENT ON <object_spec> IS <comment>

<object_spec> ::= see explanation
<comment> ::= <string_literal> | <parameter_name>
```

string_literal [Page 32], parameter_name [Page 43]

Explanation

Comments can be specified for the following database objects:

<object_spec> ::=	**Explanation**
`COLUMN <table_name>.<column_name>` table_name [Page 47], column_name [Page 46]	The column must exist in the specified table. The current user must be the owner of the table. The comment for this column can be interrogated by selecting the system table DOMAIN. COLUMNS.
`DBPROC[EDURE] <dbproc_name>` dbproc_name [Page 40]	`dbproc_name` must identify an existing database procedure [Page 28] whose owner is the current user. A comment is stored for the DB procedure. The comment can be interrogated by selecting the system table DOMAIN. DBPROCEDURES.
`DOMAIN <domain_name>` domain_name [Page 41]	`domain_name` must specify a domain [Page 24] of the current user. The comment for this domain can be interrogated by selecting the system table DOMAIN. DOMAINS.

FOREIGN KEY <table_name>.<referential_constraint_name> referential constraint name [Page 44]	referential_constraint_name must specify a referential CONSTRAINT definition [Page 128] for the specified table of the current owner. The comment for this referential CONSTRAINT definition can be interrogated by selecting the system table DOMAIN. FOREIGNKEYS.
INDEX <index_name> ON <table_name> index_name [Page 42]	index_name must specify an index [Page 24] of the specified table. The current user must be the owner of the table. The comment for this index can be interrogated by selecting the system table DOMAIN. INDEXES.
SEQUENCE <sequence_name> sequence_name [Page 45]	An existing sequence must be specified using sequence_name. The current user must be the owner of the sequence. The comment for this sequence can be interrogated by selecting the system table DOMAIN. SEQUENCES.
[PUBLIC] SYNONYM <synonym_name> synonym_name [Page 46]	synonym_name must specify a synonym [Page 24] of the current user. If PUBLIC is specified, the synonym must have the PUBLIC attribute. The comment for this synonym can be interrogated by selecting the system table DOMAIN. SYNONYMS.
TABLE <table_name> table_name [Page 47]	The specified table [Page 23] must identify a base or view table of the current user that is not a temporary table. The comment for this table can be interrogated by selecting the system table DOMAIN.TABLES.
TRIGGER <trigger_name>.<table_name> trigger_name [Page 47]	The specified trigger name must identify a trigger [Page 29] of the specified table. The current user must be the owner of the table. A comment is stored for the trigger. The comment can be interrogated by selecting the system table DOMAIN. TRIGGERS.
USER <user_name> user_name [Page 40]	The specified user [Page 25] must identify an existing user whose owner is the current user. The comment for this user can be interrogated by selecting the system table DOMAIN.USERS.
USERGROUP <usergroup_name> usergroup_name [Page 39]	The specified user group [Page 25] must identify an existing user group whose owner is the current user. The comment for this user group can be interrogated by selecting the system table DOMAIN. USERS.

<parameter_name> parameter_name [Page 43]	The corresponding variable must contain one of the values listed in the table. The values must be encapsulated in quotation marks. Example of specifying the corresponding variables: 'COLUMN <table_name>.<column_name>'

CREATE DBPROC Statement (create_dbproc_statement)

The CREATE DBPROC statement (create_dbproc_statement) defines a database procedure [Page 28].

Syntax

```
<create_dbproc_statement> ::= CREATE DBPROC <procedure_name>
[(<formal_parameter>,..)] [RETURNS CURSOR] AS <routine>

<formal_parameter> ::=
  IN <argument> <data_type>
| OUT <argument> <data_type>
| INOUT <argument> <data_type>

<argument> ::= <identifier>
```

procedure_name [Page 40], data_type [Page 119], identifier [Page 36], routine [Page 153]

The database procedure determines the average price for single rooms in hotels that are located within the specified zip code range.

```
CREATE DBPROC avg_price (IN zip CHAR(5), OUT avg_price
FIXED(6,2)) AS
    VAR sum FIXED(10,2); price FIXED(6,2); hotels INTEGER;
TRY
  SET sum = 0; SET hotels = 0;
  SELECT price FROM travel.room,travel.hotel WHERE zip =
:zip AND
  room.hno = hotel.hno AND roomtype = 'SINGLE';
    WHILE $rc = 0 DO BEGIN
      FETCH INTO :price;
      SET sum = sum + price;
      SET hotels = hotels + 1;
    END;
CATCH
  IF $rc <> 100 THEN STOP ($rc, 'unexpected error');
  IF hotels > 0 THEN SET avg_price = sum / hotels
  ELSE STOP (100, 'no hotel found');
```

Explanation

A database procedure is a subroutine that runs on the SAP DB server. SAP DB provides a language (special SQL syntax that has been extended to include variables, control structures, and troubleshooting measures) that can be used to define database procedures and triggers [Page 29].

The current user is the owner of a database procedure. He or she has the EXECUTE privilege to execute the procedure.

Parameter

When an application invokes the database procedure with the CALL statement [Page 183], it exchanges data via parameters that are defined by means of the formal parameters. A formal parameter of the database procedure usually corresponds to a variable in the application.

IN | OUT | INOUT

The parameter mode (IN | OUT | INOUT) specifies the direction in which the data is transferred when the procedure is invoked.

IN: defines an input parameter, i.e. the value of the variable is transferred to the database procedure when the procedure is invoked.

OUT: defines an output parameter, i.e. the value of the formal parameter is transferred from the database procedure to the variable after the procedure has terminated.

INOUT: defines an input/output parameter that combines the IN and OUT functions.

Argument

By specifying an argument, you assign a name to a formal parameter of the database procedure. This parameter name can then be used as a variable in expressions and assignments in the database procedure.

Data Type

Only BOOLEAN, CHAR[ACTER], DATE, FIXED, FLOAT, INT[EGER], NUMBER, REAL, SMALLINT, TIME, TIMESTAMP, and VARCHAR can be used as the data type of a formal parameter of a database procedure.

RETURNS CURSOR

If RETURNS CURSOR is specified, a database procedure is defined that returns a results table when called.

The name of this table is defined using the system variable $CURSOR. There must therefore be a statement in the database procedure that generates a results table with the result set name $CURSOR.

The value of $CURSOR is already assigned by most programming language embeddings, but can also be explicitly assigned in the database procedure.

```
CREATE DBPROC hotels_of_town (IN zip CHAR(5))
RETURNS CURSOR AS
$CURSOR = 'HOTEL_CURSOR';
DECLARE :$CURSOR CURSOR FOR
SELECT * FROM travel.hotel WHERE zip = :zip;
```

Further information

routine [Page 153]

routine

The part of the CREATE DBPROC [Page 152] or CREATE TRIGGER-statement [Page 158] referred to as the routine is the implementation of the database procedure [Page 28] or trigger [Page 29]. . It consists of optional variable declarations and statements.

Syntax

```
<routine> ::= [<local_variables>] <statement_list>;

<local_variables> ::=VAR <local_variable_list>;
<local_variable_list> ::= <local_variable> | <local_variable_list>;
<local_variable>
<local_variable> ::= <variable_name> <data_type>
<variable_name> ::= <identifier>

<statement_list> ::= <statement> | <statement_list> ; <statement>
```

identifier [Page 36], data_type [Page 119], statement [Page 154]

Explanation

Variables

The local variables of the database procedure must be declared explicitly by specifying a data type before they are used. Only BOOLEAN, CHAR[ACTER], DATE, FIXED, FLOAT, INT[EGER], NUMBER, REAL, SMALLINT, TIME, TIMESTAMP, and VARCHAR are permitted as data types data types [Page 119]. Once they have been declared, the variables can be used in any SQL and other statements.

Every database procedure has the variables $RC, $ERRMSG, and $COUNT implicitly.

The **$RC variable** returns a numeric error code after an SQL statement has been executed. The value 0 means that the SQL statement was successfully executed.
In parallel with $RC, the **$ERRMSG variable** returns an explanation of the error containing a maximum of 80 characters.
The number of lines processed in an SQL statement is indicated by the **$COUNT variable**.

Variables can be assigned a value with the `assignment_statement` (see statement [Page 154]).

Restrictions

The statement list must not contain more than 255 SQL statements.

statement

`statement` is a syntax element that is used in a routine [Page 153]. The statements specified in the `statement` syntax description can be used to define a database procedure (see CREATE DBPROC statement [Page 152]) or trigger (see CREATE TRIGGER statement [Page 158]).

Syntax

```
<statement> ::= BEGIN <statement_list> END
| BREAK | CONTINUE | CONTINUE EXECUTION
| <if_statement> | <while_statement> | <assignment_statement> |
<case_statement>
| RETURN
| STOP (<expression> [,<expression>] )
| TRY <statement_list>; CATCH <statement>
| <routine_sql_statement>

<statement_list> ::= <statement> | <statement_list> ; <statement>

<if_statement> ::= IF <search_condition> THEN <statement> [ELSE
<statement>]
<while_statement> ::= WHILE <search_condition> DO <statement>
<assignment_statement> ::= [SET] <variable_name> = <expression>
```

```
<case_statement> ::= <simple_case_statement> |
<searched_case_statement>

<routine_sql_statement> ::=
  <call_statement> | <close_statement> | <create_table_temp> |
<drop_table_temp>
| <declare_cursor_statement> | <delete_statement> | <fetch_statement>
| <insert_statement> | <lock_statement>
| <select_statement> | <named_select_statement> |
<single_select_statement>
| <subtrans_statement> | <update_statement>

<variable_name> ::= <identifier>
```

`<create_table_temp> :: = <create_table_statement>` for creating temporary tables, that is the table_name [Page 47] in the CREATE TABLE statement, must have the format `TEMP.<identifier>`.

```
<drop_table_temp> ::= DROP TABLE TEMP.<identifier>
```

expression [Page 52], search_condition [Page 70], simple_case_statement [Page 157], searched_case_statement [Page 156], call_statement [Page 183], close_statement [Page 208], create_table_statement [Page 115], declare_cursor_statement [Page 185]; delete_statement [Page 181], fetch_statement [Page 205], insert_statement [Page 173], lock_statement [Page 215], select_statement [Page 188], named_select_statement [Page 186], single_select_statement [Page 208], subtrans_statement [Page 214], update_statement [Page 178], identifier [Page 36]

Explanation

Variables specified in a routine [Page 153] can be assigned a value with the `assignment statement`.

Control Structures

The **IF statement** first evaluates the search condition [Page 70]. If this is fulfilled, the statement specified in the THEN branch is executed. Otherwise, the statement in the ELSE branch (if defined) is executed.

The **WHILE statement** enables statements to be repeated in response to certain conditions. The statement is executed as long as the specified search condition is true. The condition is checked, in particular, before the statement is executed for the first time. This means that the statement may not be executed at all. By specifying **BREAK**, you can exit the loop straight away, without checking the condition. If **CONTINUE** is specified in the loop, the condition is re-evaluated immediately and the loop is processed again or exited, depending on the result.

The **CASE statement** (`case_statement`) allows the conditional execution of a statement, dependent on search conditions or the equality of operators. There are simple and general CASE statements.

Specifying **RETURN** allows an immediate error-free of the surrounding database procedure.

Troubleshooting

If an SQL error occurs in the statement list between **TRY** and **CATCH**, the system branches directly to the statement that follows CATCH. The actual troubleshooting routine can be programmed in this statement. If **CONTINUE EXECUTE** is executed here, the system jumps directly to the point after the statement that triggered the error.

The database procedure is interrupted immediately when the **STOP** function is invoked. The value of the first parameter of the STOP function is the return or error message that the application receives as the result of the database procedure call. An error text can also be returned.

SQL Statements (routine_sql_statement)

The tables in the SQL statements of the database procedure must always be complete, that is, with the owner specified. In the case of SELECT statements, a full statement of the table name in the FROM clause [Page 196] is sufficient.

Restrictions

The statement list must not contain more than 255 SQL statements.

The length of an SQL statement (routine sql statement) must not exceed approximately 8 KB.

General CASE Statement (searched_case_statement)

The general CASE statement (`searched_case_statement`) is a syntax element that is used in a statement [Page 154]. You can use the general CASE statement to define a database procedure (see CREATE DBPROC statement [Page 152]) or a trigger (see CREATE TRIGGER statement [Page 158]).

Syntax

```
<searched_case_statement> ::= CASE
<searched_case_when_clause>...
[<case_else_clause>]
END [CASE]

<searched_case_when_clause> ::= WHEN <search_condition> THEN
<statement>
<case_else_clause> ::= ELSE <statement>
```

search_condition [Page 70], statement [Page 154]

Explanation

Variables specified in a routine [Page 153] can be assigned a value with a `statement`.

Control Structures

A **CASE statement** (`case_statement`) allows the conditional execution of a statement, dependent on search conditions or the equality of operators.

In the case of a general CASE statement (`searched_case_statement`), the first fulfilled search condition is determined, the associated statement executed, and the CASE statement ends.

```
CASE
WHEN digit = 0 THEN toCHAR = 'zero';
WHEN digit = 1 THEN toCHAR = 'one';
WHEN digit = 2 THEN toCHAR = 'two';
WHEN digit = 3 THEN toCHAR = 'three';
WHEN digit = 4 THEN toCHAR = 'four';
WHEN digit = 5 THEN toCHAR = 'five';
WHEN digit = 6 THEN toCHAR = 'six';
WHEN digit = 7 THEN toCHAR = 'seven';
WHEN digit = 8 THEN toCHAR = 'eight';
WHEN digit = 9 THEN toCHAR = 'nine';
ELSE STOP(-29000, 'no digit');
END CASE
```

In a CASE statement, if no matching literal or fulfilled search condition exists, the statement in the ELSE branch is executed.

If there is no ELSE branch, runtime error -28901 is returned.

See also:

Simple CASE statement (simple_case_statement) [Page 157]

Simple CASE Statement (simple_case_statement)

The simple CASE statement (`simple_case_statement`) is a syntax element that is used in a statement [Page 154]. You can use the simple CASE statement to define a database procedure (see CREATE DBPROC statement [Page 152]) or a trigger (see CREATE TRIGGER statement [Page 158]).

Syntax

```
<simple_case_statement> ::= CASE <expression>
<simple_case_when_clause>...
[<case_else_clause>]
END [CASE]

<simple_case_when_clause> ::= WHEN <literal>[ ...] THEN <statement>
<case_else_clause> ::= ELSE <statement>
```

expression [Page 52], literal [Page 31], statement [Page 154]

Explanation

Variables specified in a routine [Page 153] can be assigned a value with a `statement`.

Control Structures

A **CASE statement** (`case_statement`) allows the conditional execution of a statement, dependent on search conditions or the equality of operators.

In the case of a simple CASE statement (`simple_case_statement`), the expression is compared with the literals. If the expression matches a literal, the associated statement is executed and the CASE statement ends.

```
CASE digit
WHEN 0 THEN toCHAR = 'zero';
WHEN 1 THEN toCHAR = 'one';
WHEN 2 THEN toCHAR = 'two';
WHEN 3 THEN toCHAR = 'three';
WHEN 4 THEN toCHAR = 'four';
WHEN 5 THEN toCHAR = 'five';
WHEN 6 THEN toCHAR = 'six';
WHEN 7 THEN toCHAR = 'seven';
WHEN 8 THEN toCHAR = 'eight';
WHEN 9 THEN toCHAR = 'nine';
ELSE STOP(-29000, 'no digit');
END CASE
```

In a CASE statement, if no matching literal or fulfilled search condition exists, the statement in the ELSE branch is executed.

If there is no ELSE branch, runtime error -28901 is returned.

See also:

General CASE Statement (searched_case_statement) [Page 156]

DROP DBPROC statement

The DROP DBPROC statement drops a database procedure [Page 28].

Syntax

```
<drop_dbproc_statement> ::= DROP DBPROC <dbproc_name>
```

dbproc_name [Page 40]

Explanation

The specified database procedure name must identify an existing database procedure of the current user.

The metadata of the database procedure is dropped.

CREATE TRIGGER Statement (create_trigger_statement)

The CREATE TRIGGER statement defines a trigger [Page 29] for a base table (see Table [Page 23]).

Syntax

```
<create_trigger_statement> ::= CREATE TRIGGER <trigger_name> FOR
<table_name>
AFTER <trigger_event,..> EXECUTE (<routine>) [WHENEVER
<search_condition> ]

<trigger_event> :: INSERT | UPDATE [(<column_list>)] | DELETE
<column_list> ::= <column_name> | <column_list>,<column_name>
```

trigger_name [Page 47], table_name [Page 47], search_condition [Page 70], routine [Page 153], column_name [Page 46]

The trigger ensures that the hotel number in the `room` table is also changed when a hotel number is changed in the `hotel` table.

```
CREATE TRIGGER hotel_update FOR hotel AFTER UPDATE EXECUTE
(
TRY
  IF NEW.hno <> OLD.hno
  THEN UPDATE travel.room SET hno = :NEW.hno WHERE hno =
:OLD.hno;
CATCH
  IF $rc <> 100
  THEN STOP ($rc, 'unexpected error');
)
```

Explanation

A trigger is a special type of database procedure [Page 28] that is assigned to a base table. This database procedure cannot be executed explicitly with the CALL statement [Page 183], but rather automatically by SAP DB when defined events (`trigger events`) for the table occur.

SAP DB provides a language (special SQL syntax that has been extended to include variables, control structures, and troubleshooting measures) that can be used to define database procedures and triggers.

The specified synonym name must identify an existing base table of the current user.

Trigger Event

The trigger event defines what triggers the trigger. The trigger is always invoked if the triggering event has been processed correctly.

INSERT: the INSERT trigger event causes the trigger to be executed for each row inserted in the table.

UPDATE: the UPDATE event causes the trigger to be executed for each modification made to a row in the table. If a column list is specified, the trigger is only called if one of the columns in the column list was modified.

DELETE: the DELETE trigger event causes the trigger to be executed for every row deleted from the table.

A maximum of one trigger can be defined for each trigger event in each table.

Trigger Routine

Each **INSERT trigger** implicitly has a corresponding variable `NEW.<column_name>` for each column in the table. When the trigger is executed, this variable has the value of the corresponding column in the inserted row. It is only permissible to specify NEW for SQL statements specified in routine_sql_statements. Specifying NEW for the other statements leads to an error.

Each **UPDATE trigger** implicitly has a corresponding variable `NEW.<column name>` and `OLD.<column name>` for each column in the table. When the trigger is executed, the `OLD.<column_name>` variable has the value of the corresponding column in front of and `NEW.<column_name>` after the change in the row. Specifying NEW and OLD is optional.

Each **DELETE trigger** implicitly has a corresponding variable `OLD.<column_name>` for each column in the table. When the trigger is executed, this variable has the value of the corresponding column in the deleted row. It is only permissible to specify OLD for SQL statements specified in routine_sql_statements. Specifying OLD for the other statements leads to an error.

:NEW and **:OLD** must always be used with a colon in SQL statements that are used in triggers and that belong to the `routine_sql_statements` [Page 154] (For example: `UPDATE travel.room SET hno = :NEW.hno WHERE hno = :OLD.hno`).

NEW and **OLD** must always be used **without** a colon in SQL statements that are used in triggers and that do not belong to the `routine_sql_statements` (For example: `IF NEW.hno <> OLD.hno`).

See also:

routine [Page 153]

If the trigger is terminated by STOP with an error number not equal to zero, the entire SQL statement that triggered the trigger fails.

The SUBTRANS statement [Page 214] is not allowed in a trigger.

If a **WHENEVER** statement is specified, the trigger is only executed if the search condition [Page 70] is fulfilled. The condition must not contain a subquery [Page 200] nor any set function [Page 105]s.

DROP TRIGGER statement

A DROP TABLE statement deletes a trigger [Page 29] for a table.

Syntax

```
<drop_trigger_statement> ::= DROP TRIGGER <trigger_name> OF
<table_name>
```

trigger_name [Page 47], table_name [Page 47]

Explanation

The specified table name must identify an existing table of the current user.

The specified trigger name must identify an existing trigger of the table.

The metadata of the trigger is dropped.

Authorization

SQL statements for authorization

CREATE USER statement [Page 160]	DROP USER statement [Page 164]	ALTER USER statement [Page 165] RENAME USER statement [Page 167] GRANT USER statement [Page 168]
CREATE USERGROUP statement [Page 162]	DROP USERGROUP statement [Page 165]	ALTER USERGROUP statement [Page 166] RENAME USERGROUP statement [Page 168] GRANT USERGROUP statement [Page 168]
CREATE ROLE statement [Page 169]	DROP ROLE statement [Page 170]	
ALTER PASSWORD statement [Page 169]	GRANT statement [Page 170]	REVOKE statement [Page 172]

CREATE USER Statement (create_user_statement)

The CREATE USER statement (`create_user_statement`) defines a user [Page 25]. The existence and the properties of the user are recorded in the database catalog in the form of metadata.

Syntax

```
<create_user_statement> ::= CREATE USER <user_name> PASSWORD
<password>
[<user_mode>]
[TIMEOUT <unsigned_integer>] [COSTWARNING <unsigned_integer>]
[COSTLIMIT <unsigned_integer>] [[NOT] EXCLUSIVE] [DEFAULTCODE <ASCII
| EBCDIC | UNICODE>]
| CREATE USER <user_name> PASSWORD <password> LIKE <source_user>
| CREATE USER <user_name> PASSWORD <password> USERGROUP
<usergroup_name>
```

user_name [Page 40], user_mode [Page 162], unsigned_integer [Page 34], usergroup_name [Page 39]

Explanation

The current user must be a DBA user. The user is the owner of the created user.

The specified user name must not be identical to the name of an existing user, user group, or role.

The password (`password`) must be specified when an database session is started. It ensures that only authorized users can access the database system.

Unsigned integers (`unsigned_integer`) must always be greater than 0.

Unlimited disk space is available to the user for the storage of his or her private and temporary tables (in the context of the sizes specified for the data volumes during the installation).

TIMEOUT

The Timeout value is specified in seconds and must be between 30 and 32400.

Only the SYSDBA user can define users with the timeout value 0.

COSTWARNING/COSTLIMIT

COSTWARNING and COSTLIMIT limit costs by preventing users from executing QUERY statements or INSERT statements in the form of INSERT...SELECT... beyond a specified degree of complexity.

Before these SQL statements are executed, the costs expected to result from this statement are estimated. This SELECT costs estimate can be output with the EXPLAIN statement [Page 209]. In interactive mode, the estimated SELECT cost value is compared with the COSTWARNING and COSTLIMIT values specified for the user.

The COSTWARNING and COSTLIMIT values are ignored with QUERY statements or INSERT statements in the form INSERT...SELECT... which are embedded in a programming language.

COSTWARNING: specifies the estimated SELECT cost value beyond which the user receives a warning. In this case, the user is asked whether the SQL statement is to be executed.

COSTLIMIT: specifies the estimated SELECT cost value beyond which the SQL statement is not executed.

The COSTLIMIT value must be greater than the COSTWARNING value.

EXCLUSIVE

EXCLUSIVE: prevents the user from opening two different database sessions simultaneously.

NOT EXCLUSIVE: allows the user to open several database sessions simultaneously.

If the EXCLUSIVE condition is not specified, EXCLUSIVE is assumed implicitly (without NOT).

DEFAULTCODE <ASCII | EBCDIC | UNICODE>

The value of the database parameter DEFAULT_CODE is overridden with the code attribute [Page 17] specified in DEFAULTCODE for the objects of the specified user.

LIKE

The current user must have owner authorization over the source user (`source_user`).

If the source user **is not a member of a user group**, the new user receives the same user class and values for PERMLIMIT, TEMPLIMIT, TIMEOUT, COSTWARNING, COSTLIMIT, and EXCLUSIVE as the source user. The new user receives all the privileges that the source user was granted by other users.

If the source user is a **member of a user group**, a new member is created in this user group with the new user name.

USERGROUP

The user issuing the SQL statement must be the owner of the specified user group. The new user then becomes a member of this user group.

User mode

User mode is used to specify the user class or status of the defined user when a user [Page 25] is created (CREATE USER statement [Page 160]). The user class specifies the operations that the defined user can execute.

Syntax

```
<user_mode> ::= DBA | RESOURCE | STANDARD
```

Explanation

If no user class is specified, the STANDARD class is assumed implicitly.

DBA

The specified user is authorized to define private data and grant privileges for this data to other users. The user can define additional users. DBA status may only be assigned by the SYSDBA that was created when the database system was installed.

RESOURCE

The specified user is authorized to define private data and grant privileges for these objects to other users.

STANDARD

The specified user can only access private data, which was created by other users and for which he or she has the appropriate privileges, as well as view tables, synonyms, and temporary tables.

Dependencies

The user classes are hierarchically ordered as follows:

- The user class RESOURCE encompasses all the rights of STANDARD users.
- The user class DBA encompasses all the rights of RESOURCES users.
- The SYSDBA user can create DBA users. He or she has owner rights over all users. Otherwise, the SYSDBA has the same function and the same rights as a DBA user, i.e. whenever a DBA user is allowed to execute an SQL statement, this also applies to a SYSDBA user.

See also:

Users and user groups [Page 25]

usergroup_mode [Page 164]

CREATE USERGROUP Statement (create_usergroup_statement)

The CREATE USERGROUP statement (`create_usergroup_statement`) defines a user group [Page 25].

Syntax

```
<create_usergroup_statement> ::= CREATE USERGROUP <usergroup_name>
[<usergroup_mode>]
[TIMEOUT <unsigned_integer>] [COSTWARNING <unsigned_integer>]
[COSTLIMIT <unsigned_integer>] [[NOT] EXCLUSIVE] [DEFAULTCODE <ASCII
| EBCDIC | UNICODE>]
```

usergroup_name [Page 39], usergroup_mode [Page 164], unsigned_integer [Page 34]

Explanation

The current user must be a DBA user.

The specified user group name must not be identical to the name of an existing user, user group, or role.

Several users who are members of this user group can be defined using CREATE USER statement [Page 160]. All private objects created by members of the user group are identified by the user group name. The owner of a private object is the group, not the user who created the object. Each user can work with any private object of the group, as if this user were the owner of the object. Privileges can only be granted or revoked from the group. A privilege cannot be granted or revoked from a single member of the group.

Unsigned integers (`unsigned_integer`) must always be greater than 0.

Unlimited disk space is available to the user for the storage of his or her private and temporary tables (in the context of the sizes specified for the data volumes during the installation).

TIMEOUT

The Timeout value is specified in seconds and must be between 30 and 32400.

Only the SYSDBA user can define users with the timeout value 0.

COSTWARNING/COSTLIMIT

COSTWARNING and COSTLIMIT limit costs by preventing users in this user group from executing QUERY statements or INSERT statements in the form of INSERT...SELECT... beyond a specified degree of complexity and that are therefore cost-intensive.

Before these SQL statements are executed, the costs expected to result from this statement are estimated. This SELECT costs estimate can be output with the EXPLAIN statement [Page 209]. In interactive mode, the estimated SELECT cost value is compared with the COSTWARNING and COSTLIMIT values specified for the user.

The COSTWARNING and COSTLIMIT values are ignored with QUERY statements or INSERT statements in the form INSERT...SELECT... which are embedded in a programming language.

COSTWARNING: specifies the estimated SELECT cost value beyond which the user receives a warning. In this case, the user is asked whether the SQL statement is to be executed.

COSTLIMIT: specifies the estimated SELECT cost value beyond which the SQL statement is not executed.

The COSTLIMIT value must be greater than the COSTWARNING value.

EXCLUSIVE

EXCLUSIVE: prevents users in this user group from opening two different database sessions simultaneously.

NOT EXCLUSIVE: allows the user to open several database sessions simultaneously.

If the EXCLUSIVE condition is not specified, EXCLUSIVE is assumed implicitly (without NOT).

DEFAULTCODE <ASCII | EBCDIC | UNICODE>

The value of the database parameter DEFAULT_CODE is overridden with the code attribute [Page 17] specified in DEFAULTCODE for the objects of the specified user.

Usergroup name

Usergroup mode is used to specify the user class or status of the defined usergroup when a usergroup [Page 25] is created (CREATE USERGROUP statement [Page 162]).

Syntax

```
<usergroup_mode> ::= RESOURCE | STANDARD
```

Explanation

If no user class is specified, the STANDARD class is assumed implicitly.

RESOURCE

A user in the specified usergroup is authorized to define private data and grant privileges for these objects to other users.

STANDARD

A user in the specified usergroup can only access private data, which was created by other users and for which he or she has the appropriate privileges, as well as view tables, synonyms, and temporary tables.

The user class RESOURCE encompasses all the rights of STANDARD users.

See also:

Users and user groups [Page 25]

user_mode [Page 162]

DROP USER statement

A DROP USER statement drops a user [Page 25] definition. The metadata of the user to be dropped is dropped from the catalog.

Syntax

```
<drop_user_statement> ::= DROP USER <user_name> [<cascade_option>]
```

user_name [Page 40], cascade_option [Page 132]

Explanation

The current user must have owner authorization over the user to be dropped.

The specified user must not be logged onto the database system when the DROP USER statement is executed.

- If the user to be dropped does not belong to a usergroup and is the owner of synonyms or tables, and if the CASCADE option **RESTRICT** was specified, the DROP USER statement fails.
- If **no** CASCADE option or the CASCADE option **CASCADE** is specified, all the synonyms and tables of the user to be dropped, as well as all indexes, privileges, view tables, etc. based on these objects, are dropped.

Dropping a user with the user class [Page 162] DBA does not affect any users that were created by this user. The SYSDBA user then becomes the new owner of these users.

DROP USERGROUP statement

The DROP USERGROUP statement drops a usergroup [Page 25] definition. The metadata of the usergroup to be dropped is dropped from the catalog.

Syntax

```
<drop_usergroup_statement> ::= DROP USERGROUP <usergroup_name>
[<cascade_option>]
```

usergroup_name [Page 39], cascade_option [Page 132]

Explanation

The current user must have owner authorization over the usergroup to be dropped.

The users in this usergroup must not be logged onto the database system when the DROP USERGROUP statement is executed.

- If the usergroup to be dropped does not belong to a usergroup and is the owner of synonyms or tables, and if the CASCADE option **RESTRICT** was specified, the DROP USERGROUP statement fails.
- If **no** CASCADE option or the CASCADE option **CASCADE** is specified, all the synonyms and tables of the usergroup to be dropped, as well as all indexes, privileges, view tables, etc. based on these objects, are dropped.

ALTER USER Statement (alter_user_statement)

The ALTER USER statement alters the properties assigned to a user [Page 25].

Syntax

```
<alter_user_statement> ::= ALTER USER <user_name> [<user_mode>]
[TIMEOUT <unsigned_integer> | TIMEOUT NULL]
[COSTWARNING <unsigned_integer> | COSTWARNING NULL]
[COSTLIMIT <unsigned_integer> | COSTLIMIT NULL]
[DEFAULT ROLE ALL [EXCEPT <role_name>] | DEFAULT ROLE NONE
| DEFAULT ROLE <role_name> [IDENTIFIED BY <password>]]
[[NOT] EXCLUSIVE] [DEFAULTCODE <ASCII | EBCDIC | UNICODE>]
```

user_name [Page 40], user_mode [Page 162], unsigned_integer [Page 34], role_name [Page 45], password [Page 42]

Explanation

At least one of the optional clauses must be specified.

The specified user name must identify a defined user, who is not a member of a user group.

The current user must have owner authorization over the user whose properties are to be altered.

The specified user must not be logged onto the database system when the ALTER USER statement is executed.

User class (user_mode)

- **DBA**: specifies that the user is to be assigned the user class DBA. The DBA user class can only be granted by the SYSDBA.
- **RESOURCE**: specifies that the user is to be assigned the user class RESOURCE. If the user was previously assigned to the user class DBA, owner authorization for all users he or she created is revoked. The SYSDBA user then becomes the new owner.

- **STANDARD**: specifies that the user is removed from the current user class and loses the right to create base tables. All the base tables created by the user are dropped.
- **No user class**: if no user class is specified, the user class remains unchanged.

NULL

If the NULL value is specified, the value defined previously is cancelled.

DEFAULT ROLE

DEFAULT ROLE defines which of the roles [Page 25] assigned to the user is activated automatically when a database session is opened.

- **ALL**: all the roles assigned to the user are activated when a session is opened. EXCEPT can be used to exclude specified roles from activation.
- **NONE**: none of the roles is activated when a user database session is opened.
- **Role name specified**: the roles specified here must exist and be assigned to the user. They are automatically activated when a user database session is opened.

See also:

Role concept

TIMEOUT, COSTWARNING, COSTLIMIT, [NOT] EXCLUSIVE, and DEFAULTCODE are described under the CREATE USER statement [Page 160].

ALTER USERGROUP Statement (alter_usergroup_statement)

The ALTER USERGROUP statement (`alter_usergroup_statement`) alters the properties assigned to a user group [Page 25].

Syntax

```
<alter_usergroup_statement> ::= ALTER USERGROUP <usergroup_name>
[<usergroup_mode>]
[TIMEOUT <unsigned_integer> | TIMEOUT NULL]
[COSTWARNING <unsigned_integer> | COSTWARNING NULL] [COSTLIMIT
<unsigned_integer> | COSTLIMIT NULL]
[DEFAULT ROLE ALL [EXCEPT <role_name>] | DEFAULT ROLE NONE
| DEFAULT ROLE <role_name> [IDENTIFIED BY <password>]]
[[NOT] EXCLUSIVE] [DEFAULTCODE <ASCII | EBCDIC | UNICODE>]
```

usergroup_name [Page 39], usergroup_mode [Page 164], unsigned_integer [Page 34], role_name [Page 45], password [Page 42]

Explanation

At least one of the optional clauses must be specified.

The specified user group must identify a defined user group.

The current user must have owner authorization over the user group whose properties are to be altered.

The members of the specified user group must not be logged onto the database system when the ALTER USERGROUP statement is executed.

User class of the user group (usergroup mode)

- **RESOURCE**: specifies that the user group is to be assigned to the user class RESOURCE.
- **STANDARD**: specifies that the user group is removed from the current user class and loses the right to create base tables. All the base tables created by the user group are dropped.
- **No user class**: if no user class is specified, the user class remains unchanged.

NULL

If the NULL value is specified, the value defined previously is cancelled.

DEFAULT ROLE

DEFAULT ROLE defines which of the roles [Page 25] assigned to the user group is activated automatically when a session is opened by a group member.

- **ALL**: all the roles assigned to the user group are activated when a session is opened. EXCEPT can be used to exclude specified roles from activation.
- **NONE**: none of the roles is activated when a session is opened by a member of the user group.
- **Role name specified**: the roles specified here must exist and be assigned to the user group. They are automatically activated when a session of a group member is opened.

See also:

Role concept

For information about the specifications TIMEOUT, COSTWARNING, COSTLIMIT, [NOT] EXCLUSIVE, and DEFAULTCODE, see CREATE USERGROUP statement [Page 162].

RENAME USER statement

The RENAME USER statement changes the name of a user [Page 25].

Syntax

```
<rename_user_statement> ::= RENAME USER <user_name> TO
<new_user_name>
```

user_name [Page 40]

Explanation

The user to be modified must exist. The user name must identify the current user or a user over whom the current user has the owner privilege.

The new user name must not be identical to the name of an existing user, usergroup, or role.

If the name of the user to be modified is different from that of the current user, the user that is to be modified must not be logged onto the database system when the RENAME USER statement is executed. Otherwise, the current transaction is terminated with COMMIT before executing the RENAME USER statement is executed.

The database system automatically adapts the objects that are dependent on the modified user to the new user name.

RENAME USERGROUP statement

The RENAME USERGROUP statement changes the name of a usergroup [Page 25].

Syntax

```
<rename_usergroup_statement> ::= RENAME USERGROUP <usergroup_name> TO
<new_usergroup_name>
```

usergroup_name [Page 39]

Explanation

The usergroup to be modified must exist. The name of the usergroup must identify a usergroup for which the current user has the owner privilege.

The new usergroup name must not be identical to that of an existing user, usergroup, or role.

The members of this usergroup must not be logged onto the database system when the RENAME USERGROUP statement is executed.

The database system automatically adapts the objects that are dependent on the modified usergroup to the new usergroup name.

GRANT USER Statement (grant_user_statement)

The GRANT USER statement grants another user the owner privilege that the SYSDBA or a DBA user has over a user [Page 25].

Syntax

```
<grant user statement> ::= GRANT USER <granted users> [FROM
<user_name>] TO <user_name>

<granted users> ::= <user name>,...| *
```

user name [Page 40]

Explanation

The current user must be a DBA user.

The user names specified after the FROM and TO keywords must be different and must identify DBA users. If `FROM <user name>` is not specified, the current user is assumed implicitly.

The users specified after the GRANT USER keywords must exist and must not be a member of a usergroup. They must also have the user class RESOURCE or STANDARD. The FROM user must have the owner privilege for these users. If * is specified, the GRANT USER statement affects all users for which the FROM user has the owner privilege.

The FROM user grants the TO user the owner privileges which the FROM user has over the specified users. These rights are revoked from the FROM user. In particular, the TO user is granted the right to drop any specified user and to alter the user class and other properties of this user.

GRANT USERGROUP Statement (grant_usergroup_statement)

The GRANT USERGROUP statement grants the owner privilege that the SYSDBA or a DBA user has over a usergroup [Page 25] to another user.

Syntax

```
<grant_usergroup_statement> ::= GRANT USERGROUP <granted_usergroups>
[FROM <user_name>] TO <user_name>

<granted usergroups> ::= <usergroup name>,...| *
```

user name [Page 40], usergroup name [Page 39]

Explanation

The current user must be a DBA user.

The user names specified after the FROM and TO keywords must be different and must identify DBA users. If `FROM <user name>` is not specified, the current user is assumed implicitly.

The usergroup name must identify a usergroups for which the FROM user has the owner privilege. An asterisk * stands for all usergroups for which the FROM user has the owner privilege.

The FROM user grants the TO user the owner authorization which the FROM user has over the specified usergroups. These rights are revoked from the FROM user. In particular, the TO user is granted the right to drop any usergroup, to alter the user class and properties of this usergroup, and to drop or create group members.

ALTER PASSWORD statement

The ALTER PASSWORD statement is required to alter a user's password.

Syntax

```
<alter_password_statement> ::= ALTER PASSWORD <old_password> TO
<new_password>
| ALTER PASSWORD <user_name> <new_password>

<old_password> ::= <password>
<new_password> ::= <password>
```

user_name [Page 40], password [Page 42]

Explanation

The old password must match the password entered in the catalog for the current user.

If the `user_name` is specified, the current user must be the SYSDBA.

The new password must be specified in the CONNECT statement [Page 211] the next time the user starts a session.

CREATE ROLE Statement (create_role_statement)

The CREATE ROLE statement defines a role [Page 25].

Syntax

```
<create_role_statement> ::= CREATE ROLE <role_name> [IDENTIFIED BY
<password>]
```

role name [Page 45], password [Page 42]

Explanation

The current user must be a DBA user.

The role name must not be the same as the name of an existing role, user, or usergroup.

See also:

Role concept

DROP ROLE Statement (drop_role_statement)

The DROP ROLE statement drops a role [Page 25].

Syntax

```
<drop_role_statement> ::= DROP ROLE <role_name>
```

role_name [Page 45]

Explanation

The current user must be the owner of the role.

The metadata of the role to be dropped is dropped from the database catalog.

GRANT Statement (`grant_statement`)

The GRANT statement assigns privileges [Page 25] for tables, individual columns and roles, the SELECT privilege for a sequence, and the execution privilege for a database procedure.

Syntax

```
<grant_statement> ::= GRANT <priv_spec>,... TO <grantee>,... [WITH
GRANT OPTION]
| GRANT EXECUTE ON <dbproc_name> TO <grantee>,...
| GRANT SELECT ON <sequence_name> TO <grantee>,... [WITH GRANT
OPTION]
```

priv_spec [Page 171], grantee [Page 171], dbproc_name [Page 40], sequence_name [Page 45]

Explanation

The privileges in the privilege specification are assigned to the users [Page 25], user groups [Page 25], and roles [Page 25] (**see also:** Role Concept) specified in the grantee list.

WITH GRANT OPTION

Users or usergroups identified as grantees are allowed to pass on their privileges to other users. The current user must have the authorization to pass on these privileges.

The WITH GRANT OPTION cannot be specified if `grantee` identifies a role.

GRANT EXECUTE ON

GRANT EXECUTE ON allow the user identified by `grantee` to execute the specified database procedure [Page 28]. The current user must be the owner of the database procedure.

GRANT SELECT ON

GRANT SELECT ON allows the user identified by `grantee` to execute the specified sequence.

Privilege specification (priv_spec)

A privilege specification (`priv_spec`) defines a role or a set of privileges [Page 25] for specific tables.

Syntax

```
<priv_spec> ::= ALL [PRIV[ILEGES]] ON [TABLE] <table_name>,...
| <privilege>,... ON [TABLE] <table_name>,... | <role_name>
```

table_name [Page 47], privilege [Page 43], role_name [Page 45]

Explanation

These tables must not be temporary base tables.

The user [Page 25] must have the authorization to grant (GRANT statement [Page 170]) and revoke (REVOKE statement [Page 172]).privileges for the specified tables. For base tables, the owner of the table has this authorization.

In the case of view tables (see Table [Page 23]), the owner may not always be authorized to assign or revoke all privileges. The database determines the privileges that a user can assign or revoke for a view table when the table is created. The result depends on the type of table and on the user's privileges for the tables selected in the view table. The owner of a table can interrogate the privileges that he or she is allowed to grant or revoke by selecting the system table DOMAIN.PRIVILEGES.

A list of all the privileges that can be granted is provided in the privilege type [Page 43].

If a role is defined as a privilege specification, it must exist and the current user must be the owner of the role.

ALL [PRIV[ILEGES]]

All of the privileges that the user can grant for tables are granted (GRANT statement) or revoked (REVOKE statement) for the specified users, usergroups, and roles.

If a user who is not the owner of the table specifies ALL in a REVOKE statement, all of the privileges he or she has granted to the specified user for this table are revoked.

grantee

In `grantee` the user [Page 25] (or several users) or role [Page 25] is specified for which privileges are to be granted with the GRANT statement [Page 170] or revoked with the REVOKE statement [Page 172].

Syntax

```
<grantee> ::= PUBLIC | <user_name> | <usergroup_name> | <role_name>
```

user_name [Page 40], usergroup_name [Page 39], role_name [Page 45]

Explanation

A user in the `grantee` list must not be identical to the user name of the current user or the name of the owner of the table. A user in the `grantee` list must not denote a member of a usergroup [Page 25].

Roles

If a role is assigned to a user or usergroup, it extends the set of roles which can be activated for this user or usergroup. The user activates the role either with the SET statement [Page 212] or by including the role in the set of roles automatically activated when a session was opened with the ALTER USER statement [Page 165] or ALTER USERGROUP statement [Page 166].

A cycle may not be created when a role is assigned to a role, that is

- a role may not be assigned to itself.
- if a role R1 is assigned to a role R2, R2 may not be assigned to R1.
- if a role R1 is assigned to a role R2 and R2 is assigned to a role R3, R3 may not be assigned to either R2 or R1.
- etc.

PUBLIC

The listed privileges are granted to all users, both to current ones and to any created later.

A role cannot be assigned to PUBLIC.

REVOKE Statement (revoke_statement)

The REVOKE statement revokes privileges [Page 25].

Syntax

```
<revoke statement> ::= REVOKE <priv spec>,... FROM <grantee>,...
[<cascade_option>]
| REVOKE EXECUTE ON <dbproc_name> FROM <grantee>,...
| REVOKE SELECT ON <sequence_name> FROM <grantee>,...
[<cascade_option>]
```

priv_spec [Page 171], grantee [Page 171], cascade_option [Page 132], dbproc_name [Page 40], sequence_name [Page 45]

Explanation

The owner of a table can revoke the privileges granted for this table from any user [Page 25].

If a user is not the owner of the table, he may only revoke the privileges he has granted.

If the SELECT privilege was granted for a table any specified column names, REVOKE SELECT (<column name>,...) can be used to revoke the SELECT privilege (see privilege type [Page 43]) for the specified columns; the SELECT privilege for table columns that have not been specified remains unchanged. The same is true for the UPDATE, REFERENCES, and SELUPD privileges.

The REVOKE statement can cascade; that is, revoking a privilege from one user can result in this privilege being revoked from other users who have received the privilege from the user in question.

Let U1, U2, and U3 be users.
U1 grants U2 the privilege set P WITH GRANT OPTION.
U1 grants U3 the privilege set P' (P'<=P).

If U1 revokes the privilege set P'' (P''<=P) from user U2, the privilege set (P'*P'') is revoked implicitly from user U3.

- Whenever the SELECT privilege is revoked from the owner of a view table for a selected that does not occur in the `table_expression` of the view definition (CREATE VIEW statement [Page 143]), the column defined by `select_column` is dropped from the view table.
  If this view table is used in the FROM clause [Page 196] of another view table, the described procedure is applied recursively to this view table.
- If the SELECT privilege is revoked from the owner of a view table for a column or table occurring in the `table_expression` of the view definition, the view table is dropped, along with all view tables (see Table [Page 23]), privileges [Page 25], and synonyms

[Page 24] based on this view table, if no CASCADE option [Page 132] or the CASCADE option CASCADE is specified. The REVOKE statement will fail if the CASCADE option RESTRICT is specified.

REVOKE EXECUTE

If REVOKE EXECUTE is specified, the authorization to execute the database procedure [Page 28] is revoked from the user identified by `grantee`. The authorization for execution can only be revoked by the owner of the database procedure.

Data Manipulation

The following sections provide an introduction to the Data Manipulation Language (DML) used by the database system.

Every SQL statement that manipulates data implicitly sets an exclusive lock (see also Lock Behavior) for each inserted, updated, or deleted row.

SQL statements for data manipulation

INSERT statement [Page 173]	UPDATE statement [Page 178]	DELETE statement [Page 181]
NEXT STAMP statement [Page 183]	CALL statement [Page 183]	

INSERT Statement (insert_statement)

The INSERT statement creates new rows in a table [Page 23].

Syntax

```
<insert_statement> ::=
  INSERT [INTO] <table_name> [(<column_name>,...)] VALUES
(<insert_expression>,...) [<duplicates_clause>]
| INSERT [INTO] <table_name> [(<column_name>,...)] <query_expression>
[<duplicates_clause>]
| INSERT [INTO] <table_name> SET <set_insert_clause>,...
[<duplicates_clause>]

<insert_expression> ::= <extended_expression> | <subquery>
```

table_name [Page 47], column_name [Page 46], extended_expression [Page 177], duplicates_clause [Page 176], subquery [Page 200], query_expression [Page 189], set_insert_clause [Page 178]

Explanation

The table name must denote an existing base table, view table (see Table [Page 23]), or a synonym [Page 24].

If column names or a SET INSERT clause are specified, all of the specified column names must be columns in the table. If the table was defined without a key (that is, if the SYSKEY column is created internally by the database), the SYSKEY column must not occur in the sequence of column names or in a SET INSERT clause. A column must not occur more than once in a sequence of column names or in more than one SET INSERT clause.

The user must have the INSERT privilege for the table identified by the table name. If the table name identifies a view table, even the owner of the view table may not have the INSERT privilege because the view table cannot be changed.

A specified column (identified by `column name` or the column name in the `set_insert_clause`) is a **target column**. Target columns can be specified in any order.

- If you do not specify a column name and SET INSERT clause, this is the same as specifying a sequence of columns containing all the columns in the table in the sequence specified in the CREATE TABLE statement [Page 115] or CREATE VIEW statement [Page 143]. In this case, every table column defined by the user is a target column.
- The number of expressions (`extended_expressions`) must be equal to the number of target columns. The i^{th} expression is assigned to the i^{th} column name.
- Expressions (`extended_expression`) and subqueries (`subquery`) can be specified simultaneously.
- You can enter one or more subqueries, but no more than 64 subqueries.
- Values for any number of target columns can be supplied in a subquery.
- The specified subqueries may only supply a single result line.
- The number of selected columns [Page 193] specified in the QUERY expression must be identical to the number of target columns.
- All mandatory columns of the table identified by table name must be target columns.
- If the table name identifies a view table, rows are inserted the base table(s) on which the view table is based. In this case, the target columns of the specified table name correspond to columns of the base tables, on which the view table is based. In the following paragraphs, the term target column always refers to the corresponding column in the base table.

Further information

- Data type of the target column and data type of the value to be inserted [Page 174]
- Join view table in INSERT statement [Page 175]
- QUERY expression in INSERT statement [Page 175]
- DUPLICATES clause [Page 176]
- Constraint definition in INSERT statement [Page 177]
- Trigger in INSERT statement [Page 177]

In the case of the INSERT statement, the third entry of SQLERRD in the SQLCA is set to the number of inserted rows.

If errors occur while inserting rows, the INSERT statement fails, leaving the table unmodified.

Data type of the target column and inserted value

Let C be a target column and v a value that is not equal to the NULL value.

v is to be inserted in C by means of an INSERT statement [Page 173].
v is to be used to modify C with an UPDATE statement [Page 178].

Target column C	Required value for v

Numeric column	Number within the permissible range of C. INSERT statement: if v is the result of a QUERY expression, decimal places are rounded off if necessary. UPDATE statement: if v is the result of an expression that does not comprise one single numeric literal [Page 33], decimal places are rounded off if necessary.
Alphanumeric column with the code attribute ASCII or EBCDIC	Character string [Page 16] whose length is not greater than the length attribute of C. Subsequent blanks are ignored when the length of v is calculated. If the length of v is shorter than the length attribute of C, then v is lengthened by the corresponding number of blanks. When assigning an alphanumeric value with the code attribute ASCII (EBCDIC) to a column with the code attribute EBCDIC (ASCII), the value is implicitly converted before it is assigned.
Alphanumeric column with the code attribute BYTE	Hexadecimal character string whose length is not greater than the length attribute of C. Subsequent binary zeros are ignored when the length of v is calculated. If the length of v is shorter than the length attribute of C, then v is lengthened by the corresponding number of binary zeros.
Data type DATE	Date value [Page 17] in the current date format.
Data type TIME	Time value [Page 17] in the current time format.
Data type TIMESTAMP	Timestamp value [Page 17] in the current timestamp format.
Data type BOOLEAN	One of the values TRUE or FALSE (BOOLEAN [Page 17])

Join View Table in INSERT Statement

If the table name does **not identify a join view table** in an INSERT statement [Page 173] (see CREATE VIEW statement [Page 143]), and a row containing the key of the row to be inserted already exists, the result will depend on the DUPLICATES clause [Page 176]. The INSERT statement will fail if no DUPLICATES clause is specified.

If the table name identifies a **join view table**, a row is inserted into each base table on which the view table is based. If the key table of the view table already contains a row with the key of the row to be inserted, the INSERT statement will fail. If any row in a base table, which is not the key table of the view table, already contains the key of the row to be inserted, the INSERT statement will fail if the row to be inserted does not match the existing row.

QUERY Expression in INSERT Statement

If a QUERY expression [Page 189] is specified in the INSERT statement [Page 173] , the specified table must not be a join view table.

The QUERY expression defines a result table (see result table name [Page 41]) whose i^{th} column is assigned to the i^{th} target column. A row is formed from each row in the result table and inserted in the base table. Each of these rows has the following contents:

- Each base table column that is a target column of the INSERT statement contains the value of the corresponding column in the current result table row.

If a **QUERY expression is not specified** in the INSERT statement, exactly one row is inserted in the specified table. The inserted row has the following contents:

- Each base table column that is a target column of the INSERT statement contains the value assigned to the respective target column.

The following still applies to the inserted row(s):

- All columns of the base table that are not target columns of the INSERT statement and for which a DEFAULT specification [Page 125] exists contain the DEFAULT value.
- All columns of the base table that are not target columns of the INSERT statement and for which no DEFAULT specification exists contain the NULL value [Page 15].

DUPLICATES clause

The DUPLICATES clause can be used to determine how key collisions are handled.

Syntax

```
<duplicates_clause> ::= REJECT DUPLICATES | IGNORE DUPLICATES |
UPDATE DUPLICATES
```

Explanation

SQL statements in which the DUPLICATES clause is used

CREATE TABLE statement [Page 115]

`REJECT DUPLICATES` or no DUPLICATES clause	The CREATE TABLE statement fails if key collisions occur.
`IGNORE DUPLICATES`	Any rows that key collisions on insertion are ignored.
`UPDATE DUPLICATES`	Any rows that key collisions on insertion overwrite the rows with which they collide.

INSERT statement [Page 173]

If there is already a row in the base table with the key of the row to be inserted, the following cases must be distinguished:

`REJECT DUPLICATES` or no DUPLICATES clause	The INSERT statement fails.
`IGNORE DUPLICATES`	The new row is not inserted and processing of the INSERT statement is continued.
`UPDATE DUPLICATES`	The existing row is overwritten by the new row and processing of the INSERT statement is continued.

If there is more than one key collision for the same key for an INSERT statement with UPDATE DUPLICATES and QUERY expression specification, it is impossible to predict the content of the respective base table row once the INSERT statement has been completed.

If, for an INSERT statement with IGNORE DUPLICATES and a QUERY expression, more than one row in the result table produces the same base table key, and if this key did not exist before in the base table, it is impossible to predict the row that will be inserted in the table.

If the table name specified in the INSERT statement identifies a table without a user-defined key, the DUPLICATES clause has no effect.

Constraint Definition in INSERT Statement

If CONSTRAINT definitions [Page 127] exist for base tables in which rows are to be inserted with the INSERT statement [Page 173], each row that is to be inserted is checked against the CONSTRAINT definition. The INSERT statement fails if this is not the case for at least one row.

If at least one of the base tables in which rows are to be inserted with the INSERT statement is the referencing table of a referential CONSTRAINT definition [Page 128], the database system checks each row to be inserted to determine whether the foreign key resulting from the row exists as a key or as a value of an index defined with UNIQUE (see CREATE INDEX statement [Page 148]) in the corresponding referenced table. The INSERT statement fails if this is not the case for at least one row.

Trigger in INSERT Statement

If triggers [Page 29] that are to be executed after an INSERT statement [Page 173] were defined for base tables in which rows are to be inserted with the INSERT statement, they are executed accordingly. The INSERT statement will fail if one of these triggers fails.

Extended expression

An extended expression can be specified by means of an expression [Page 52] or one of the keywords DEFAULT or STAMP.

Syntax

```
<extended_expression> ::= <expression> | DEFAULT | STAMP
```

expression [Page 52]

Explanation

- Expression
  INSERT statement [Page 173]: an expression in an INSERT statement must not contain a column specification [Page 47].
  The value specified by a parameter specification [Page 48] in an expression is the value of the parameter identified by the specification. If an indicator parameter is specified with a negative value, the value defined by the parameter specification [Page 48] is a NULL value.
- Keyword DEFAULT
  DEFAULT denotes the value used as the DEFAULT for the column.
- STAMP key word
  The database system is able to generate unique values. This is a serial number that starts with X'000000000001'. The values are assigned in ascending order. It cannot be ensured that a sequence of values is uninterrupted. The STAMP key word supplies the next value generated by the database system.
  The STAMP keyword can be used in the INSERT statement [Page 173] or in the UPDATE statement [Page 178], but only for columns of the data type CHAR(n) BYTE with n>=8 (default specification [Page 125]).
  If the user wants to find out the generated value before it is applied to a column, he or

she must use the following **SQL statement**:
NEXT STAMP statement [Page 183]

See also:

SET INSERT condition [Page 178]

SET UPDATE condition [Page 180]

SET INSERT clause

SET INSERT clause

Syntax

```
<set_insert_clause> ::= <column_name> = <extended_value_spec>
```

column_name [Page 46], extended_value_spec [Page 49]

Explanation

SQL statements in which the SET INSERT clause is used

INSERT statement [Page 173]

UPDATE Statement

The UPDATE statement changes column values in table rows.

Syntax

```
<update_statement> ::=
  UPDATE [OF] <table_name> [<reference_name>] SET
<set_update_clause>,... [KEY <key_spec>,...] [WHERE
<search_condition>]
| UPDATE [OF] <table_name> [<reference_name>] (<column_name>,...)
VALUES (<extended_value_spec>,...) [KEY <key_spec>,...] [WHERE
<search_condition>]
| UPDATE [OF] <table_name> [<reference_name>] SET
<set_update_clause>,... WHERE CURRENT OF <result_table_name>
| UPDATE [OF] <table_name> [<reference_name>] (<column_name>,...)
VALUES (<extended_value_spec>,...) WHERE CURRENT OF
<result_table_name>
```

table name [Page 47], reference name [Page 45], set update clause [Page 180], key spec [Page 52], search condition [Page 70], column name [Page 46], extended value spec [Page 49], result table name [Page 41]

Explanation

The table name must denote an existing base table, view table (see Table [Page 23]), or a synonym [Page 24].

Columns whose values are to be updated are called **target columns**. One or more target columns and new values for these columns are specified after the table name and reference name (if necessary).

- All target columns must identify columns in the specified table, and each target column may only be listed once.
- The number of specified values (extended_value_spec [Page 49]) must be identical to the number of target columns. The i^{th} value specification is assigned to the i^{th} target column.

- The current user must have the UPDATE privilege for each target column in the specified table.
  If the table name identifies a view table, even the owner of the view table may not be able to update column values because the view table is not updateable.
- If the specified table is a view table, column values are only updated in rows in the base tables on which the view table is based. In this case, the target columns of the specified table correspond to columns in the base tables on which the view table is based. In the following paragraphs, the term target column always refers to the corresponding column in the base table.

Data type of the target column and data type of the value to be inserted [Page 174]

The following specifications determine the rows in the table that are updated:

- Optional sequence of key specifications [Page 52] and optional search condition [Page 70]
  Key specification and no search condition: a row with the specified key values already exists. The corresponding values are assigned to the target columns in this row. No rows are updated if a row with the specified key values does not exist.
  Key specification and a search condition: a row containing the specified key values exists. The search condition is applied to this row. If the search condition is satisfied, the corresponding values are assigned to the target columns of this row. No rows are updated if a row with the specified key values does not exist or if a search condition applied to a row is not fulfilled.
  No key specification and a search condition: the search condition is applied to each row in the specified table. The corresponding values are assigned to the target columns of all rows that satisfy the search condition.
- If **CURRENT OF** is used, i.e. if the cursor position in the result table [Page 41] (`result_table_name`) is specified: no rows are updated if the cursor is not positioned on a row in the result table.
- If none of the above specifications were made, all of the rows in the specified table are updated.
- If no row is found that satisfies the conditions defined by the optional clauses, the following message appears: `100 row not found`.

Even values in key columns that were defined by a user in a CREATE TABLE statement [Page 115] or ALTER TABLE statement [Page 133] can be updated. The implicit key column SYSKEY, if created, cannot be updated.

If the table name specifies a join view table (see CREATE VIEW statement [Page 143]), columns may exist that can only be updated in conjunction with other columns (determine the column combination for a column or a join view table [Page 181]). To update the value of the relevant column, a value must be specified for all of the columns in the column combination. This is true of all target columns that fulfill the following conditions:

- The target columns are located in a base table that is not a key table of the join view table and does not have a 1 : 1 relationship with the key table of the join view table.
- The target columns are foreign key columns of a referential CONSTRAINT definition [Page 128] that is relevant for the join view table.

CURRENT OF

If CURRENT OF is specified, the table name in the FROM clause [Page 196] of the QUERY statement, with which the result table was built, must be identical to the table name in the UPDATE statement.

If CURRENT OF is specified and the cursor is positioned on a row in the result table, the corresponding values are assigned to the target columns of the corresponding row. The corresponding row is the row of the table specified in the FROM clause of the QUERY statement, from which the result table row was formed. This procedure requires that the result table was specified with FOR UPDATE. It is impossible to predict whether or not the updated

values in the corresponding row are visible the next time the same row of the result table is accessed.

Reasons for an UPDATE statement failure

If CONSTRAINT definitions [Page 127] exist for base tables in which rows were updated with the UPDATE statement, each row that was updated is checked against the CONSTRAINT definitions. The UPDATE statement fails if this is not the case for at least one modified row.

For each row in which the value of foreign key columns has been updated with the UPDATE statement, the database system checks whether the respective resulting foreign key exists as a key or as a value of an index defined with UNIQUE (see CREATE INDEX statement [Page 148]) in the corresponding referenced table. The UPDATE statement fails if this is the case for at least one modified row.

For each row in which the value of a referenced column of a referential CONSTRAINT definition [Page 128] is to be updated using the UPDATE statement, the database system checks whether there are rows in the corresponding foreign key table that contain the old column values as foreign keys. The UPDATE statement fails if this is the case for at least one row.

If triggers [Page 29] that are to be executed after an UPDATE statement were defined for base tables in which rows are to be updated with the UPDATE statement, they are executed accordingly. The UPDATE statement will fail if one of these triggers fails.

Further information

The update statement can only be used to assign a value to columns with the data type LONG if it contains a parameter or NULL specification. The assignment of values to LONG columns is therefore only possible with some database tools.

In the case of the UPDATE statement, the third entry of SQLERRD in the SQLCA is set to the number of updated rows. Rows are also counted as updated when the old value was overwritten with a new but identical value.

If errors occur while rows are updated, the UPDATE statement fails, leaving the table unchanged.

SET UPDATE clause

SET UPDATE clause

Syntax

```
<set_update_clause> ::= <column_name> = <extended_expression>
| <column_name>,... = (<extended_expression>,...)
| (<column_name>,...) = (<extended_expression>,...)
| <column_name> = <subquery>
| (<column_name>,...) = <subquery>
```

column_name [Page 46], extended_expression [Page 177], subquery [Page 200]

Explanation

The expression of a SET UPDATE clause must not contain a set function [Page 105].

The subquery must produce a result table with at most one row. The number of columns must be equal to the number of target columns specified.

SQL statement in which the SET UPDATE clause is used

UPDATE statement [Page 178]

Column combination for a given column of a join view table

To determine the combination of columns for a given column v in the join view table V, use the following procedure:

1. Determine the base table T_j containing the column which corresponds to v.
2. Determine the unique table sequence Til...Tik that contains Tj.
3. Determine the last table Til in this sequence that has a 1:1 relationship with the key table.
4. The columns of V, which correspond to the foreign key columns of Til of the referential CONSTRAINT definition between Til and Til+I that is relevant for V, are elements of the column combination.
5. All columns of V which correspond to columns of the tables Til+I...Tik are elements of the column combination.

To update the column value of the column v in an UPDATE statement [Page 178], a value must be specified for each of the columns of the column combination.

DELETE statement

The DELETE statement deletes rows in a table.

Syntax

```
<delete_statement> ::=
  DELETE [FROM] <table_name> [<reference_name>] [KEY <key_spec>,...]
[WHERE <search_condition>]
| DELETE [FROM] <table_name> [<reference_name>] WHERE CURRENT OF
<result_table_name>
```

table name [Page 47], reference name [Page 45], key spec [Page 52], search condition [Page 70], result table name [Page 41]

Explanation

The table name must denote an existing base table, view table (see Table [Page 23]), or a synonym [Page 24].

The current user must have the DELETE privilege for the specified table. If the table name identifies a view table, even the owner of the view table may not have the DELETE privilege because the view table cannot be changed.

Table name identifies a view table: the rows of the tables on which the view tables are based are deleted.

Table name identifies a join view table: only the following rows are deleted:

- Rows in the key table of the join view table
- Rows in base tables on which the view table is based and that have a 1:1: relationship with the key table.

The following specifications determine the rows in the table that are deleted:

- Optional sequence of key specifications [Page 52] and optional search condition [Page 70]
  Key specification and no search condition: a row with the specified key values already exists. This row is deleted. No rows are deleted if a row with the specified key values does not exist.
  Key specification and a search condition: a row containing the specified key values exists. The search condition is applied to this row. If the search condition is satisfied,

then the row is deleted. No rows are deleted if a row with the specified key values does not exist or if a search condition applied to a row is not fulfilled.
No key specification and a search condition: the search condition is applied to each row in the specified table. All rows for which the search condition is satisfied are deleted.

- If **CURRENT OF** is used, i.e. if the cursor position in the result table [Page 41] (`result_table_name`) is specified: no rows are deleted if the cursor is not positioned on a row in the result table.
- If none of the above specifications were made, all of the rows in the specified table are deleted.
- If no row is found that satisfies the conditions defined by the optional clauses, the following message appears: `100 row not found`.

CURRENT OF

If CURRENT OF is specified, the table name in the FROM clause [Page 196] of the QUERY statement, with which the result table was built, must be identical to the table name in the DELETE statement.

If CURRENT OF is specified and the cursor is positioned on a row in the result table, the corresponding row is deleted. The corresponding row is the row of the table specified in the FROM clause of the QUERY statement, from which the result table row was formed. This procedure requires that the result table was specified with FOR UPDATE. Afterwards, the cursor is positioned behind the result table row. It is impossible to predict whether or not the updated values in the corresponding row are visible the next time the same row of the result table is accessed.

DELETE rule

For each row deleted in the course of the DELETE statement which originates from a referenced table of at least one referential CONSTRAINT definition [Page 128], one of the following actions is carried out - depending on the DELETE rule [Page 130] of the referential constraint definition:

- DELETE CASCADE: all matching rows in the corresponding foreign key table are deleted.
- DELETE RESTRICT: if there are matching rows in the corresponding foreign key table, the DELETE statement fails.
- DELETE SET NULL: the NULL value is assigned to the respective foreign key columns of all matching rows in the corresponding foreign key table.
- DELETE SET DEFAULT: the DEFAULT value that was set with a DEFAULT specification [Page 125] is assigned to the respective foreign key columns of all matching rows in the corresponding foreign key table.

Trigger

If triggers [Page 29] that are to be executed after a DELETE statement were defined for base tables from which rows are to be deleted with the DELETE statement, they are executed accordingly. The DELETE statement will fail if one of these triggers fails.

Further information

In the case of the DELETE statement, the third entry of SQLERRD in the SQLCA is set to the number of deleted rows. If this counter has the value -1, either a significant part of the table or the entire table was deleted by the DELETE statement.

If errors occur in the course of the DELETE statement, the statement fails, leaving the table unchanged.

NEXT STAMP statement

The NEXT STAMP statement supplies a unique key that was generated by the database system.

Syntax

```
<next_stamp_statement> ::= NEXT STAMP [INTO] <parameter_name>
```

parameter_name [Page 43]

Explanation

The database system is able to generate unique values. This is a serial number that starts with X'0000000000001'. The values are assigned in ascending order. It cannot be ensured that a sequence of values is uninterrupted. These values can be stored in a column with the data type CHAR [Page 120](n) BYTE with n>=8.

The NEXT STAMP statement assigns the next key generated by the database system to the variable denoted by `parameter name`.

The NEXT STAMP statement can only be embedded in a programming language and cannot be used in interactive mode.

The keyword STAMP can be also used in an INSERT statement [Page 173] or an UPDATE statement [Page 178] if the next value is to be generated by the database system and stored in a column without the user knowing the value.

CALL Statement (call_statement)

The CALL statement causes a database procedure [Page 28] to be executed.

Syntax

```
<call_statement> ::= CALL <dbproc_name> [(<expression>,...)] [WITH
COMMIT]
```

dbproc name [Page 40], expression [Page 52]

Explanation

The specified database procedure name must identify an existing database procedure.

The current user must have the EXECUTE privilege for the database procedure.

The number of expressions must be equal to the number of formal parameters for the database procedure.

The data type of the i^{th} expression must be compatible with the data type of the i^{th} formal parameter for the database procedure.

If the i^{th} formal parameter for the database procedure has the OUT or INOUT mode (see CREATE DBPROC statement [Page 152]), the corresponding expression must be a parameter specification [Page 48].

WITH COMMIT

If WITH COMMIT is specified, the current transaction [Page 210] is terminated with COMMIT after the database procedure has been executed correctly. If execution of the database procedure fails, the current transaction is terminated with ROLLBACK.

Data Query

The following sections provide an introduction to the query language (also referred to as the data retrieval language) used by the database system.

SQL statements for data queries

QUERY statement [Page 184]	SINGLE SELECT statement [Page 208]	EXPLAIN statement [Page 209]
OPEN CURSOR statement [Page 204]	FETCH statement [Page 205]	CLOSE statement [Page 208]

QUERY statement

A QUERY statement specifies a result table that can be ordered (see result table name [Page 41]). There are four different ways of formulating a QUERY statement.

Syntax

```
<query_statement> ::= <declare_cursor_statement> |
<recursive_declare_cursor_statement>
| <named_select_statement> | <select_statement>
```

declare_cursor_statement [Page 185], recursive_declare_cursor_statement [Page 186], named_select_statement [Page 186], select_statement [Page 188]

Explanation

A QUERY statement generates a named/unnamed result table [Page 185].

A distinction is made between the following QUERY statements:

Basic types of QUERY statement	
DECLARE CURSOR statement	A named result table is defined. The table is generated with an OPEN CURSOR statement [Page 204].
Recursive DECLARE CURSOR statement	This statement can be used to generate bills of material.
SELECT statement (`named_select_statement`)	A named result table is defined and generated.
SELECT statement (`select_statement`)	An unnamed result table is defined and generated.

The SELECT statement (`named_select_statement`) and SELECT statement (`select_statement`) are subject to the rules that were specified for the DECLARE CURSOR statement [Page 185] and those that were specified for the OPEN CURSOR statement.

The order of rows in the result table depends on the internal search strategies of the system and is arbitrary. The only reliable means of sorting the result rows is to specify an ORDER clause [Page 202].

Updateable result table

A result table or the underlying base tables are updateable if the QUERY statement satisfies the following conditions:

- See the section entitled "Updateable result table" in the SELECT statement (named_select_statement) [Page 186] or SELECT statement (select_statement) [Page 188]
- The result table is a named result table, i.e. it must not have been generated by a SELECT statement (`select_statement`).

Named/Unnamed Result Table

The difference between a named result table and an unnamed result table (see result table name [Page 41]) is that the unnamed result table cannot be specified in a FROM clause [Page 196] or in CURRENT OF <`result_table_name`> of a subsequent SQL statement.

QUERY statement	
DECLARE CURSOR statement (declare_cursor_statement) [Page 185]	A named result table is generated. The column names of a result table defined by this QUERY statement do not have to be unique.
SELECT statement (select_statement) [Page 188]	A unnamed result table is generated. The column names of a result table defined by this QUERY statement do not have to be unique.
SELECT Statement (named_select_statement) [Page 186]	A named result table is generated. The column names of a result table generated by this QUERY statement must be unique.

DECLARE CURSOR statement

The DECLARE CURSOR statement defines a named result table (see named/unnamed result table [Page 185]) with the name `result_table_name`.

Syntax

```
<declare_cursor_statement> ::= DECLARE <result_table_name> CURSOR FOR
<select_statement>
```

result_table_name [Page 41], select_statement [Page 188]

Explanation

An OPEN CURSOR statement [Page 204] with the name of the result table is required to actually generate the result table defined with a DECLARE CURSOR statement.

See also:

SELECT statement (select_statement) [Page 188]

Recursive DECLARE CURSOR statement

The recursive DECLARE CURSOR statement can be used to receive bills of material by means of a command.

Syntax

```
<recursive_declare_cursor_statement> ::= DECLARE <result_table_name>
CURSOR FOR
WITH RECURSIVE <reference_name> (<alias_name>,...) AS
(<initial_select> UNION ALL <recursive_select>) <final_select>

<initial_select> ::= <query_spec>
<recursive_select> ::= <query_spec>
<final_select> ::= <select_statement>
```

result_table_name [Page 41], reference_name [Page 45], alias_name [Page 39], query_spec [Page 192], select_statement [Page 188]

```
DECLARE C CURSOR FOR
WITH RECURSIVE PX (MAJOR, MINOR, NUMBER, MAINMAJOR) AS
  (SELECT W,X,Y,W FROM T WHERE W = 'aaa' UNION ALL
   SELECT W,X,Y,MAINMAJOR FROM T, PX WHERE MINOR = T.W)
 SELECT MAINMAJOR,MINOR,NUMBER FROM PX ORDER BY NUMBER
```

Explanation

- The QUERY specification [Page 192] **`initial_select`** is executed and the result is entered in a temporary result table whose name is defined by specifying the reference name. The column names contained in it receive the names from the list of alias names. The number of output columns in the QUERY specification must be identical to the number of alias names.
- The QUERY specification **`recursive select`** should comprise a SELECT statement that contains at least the reference name in the FROM clause [Page 196] and one JOIN predicate [Page 62] between this table and a different table from the FROM clause.
  The QUERY specification `recursive_select` is repeated until it does not produce a result. The respective results are (logically) entered in the temporary result table whose name is defined by the reference name. This table is extended continuously. It is ensured, however, that the results of the n^{th} execution are used for the $n+1^{th}$ execution to avoid an endless loop.
- The SELECT statement `final_select` must only contain one QUERY expression that comprises a QUERY specification [Page 192].
  This is a SELECT statement across the table with the specified reference name in which the following elements can be used: set function names [Page 108], GROUP clause [Page 199], HAVING clause [Page 200], ORDER clause [Page 202], LOCK option [Page 203]

If a result table name [Page 41] with the specified reference name existed before the recursive DECLARE CURSOR statement was executed, the corresponding cursor is closed implicitly.

SELECT Statement (named_select_statement)

A SELECT statement (`named_select_statement`) defines and creates a result table with the name `result_table_name` (see named/unnamed result table [Page 185]).

Syntax

```
<named_select_statement> ::= <named_query_expression>
[<order_clause>] [<update_clause>] [<lock_option>] [FOR REUSE]
```

named query expression [Page 191], order clause [Page 202], update clause [Page 203], lock option [Page 203]

Explanation

An OPEN CURSOR statement [Page 204] is not permitted for result tables created with this SELECT statement.

The SELECT statement (`named select statement`) is subject to the rules that were specified for the DECLARE CURSOR statement [Page 185] and those that were specified for the OPEN CURSOR statement.

Depending on the search strategy, either all the rows in the result table are searched when the SELECT statement (`named select statement`) is executed and the result table is physically generated, or each next result table row is searched when a FETCH statement [Page 205] is executed, without being physically stored. This must be taken into account for the time behavior of FETCH statements.

Updateable result table

A result table or the underlying base tables are updateable if the QUERY statement satisfies the following conditions:

- The QUERY expression (`named_query_expression`) must only comprise one QUERY specification (named_query_spec) [Page 195].
- Only one base table or one updateable view table may be specified in the FROM clause [Page 196] of the QUERY specification (`named query spec`).
- The DISTINCT keyword (see DISTINCT specification [Page 193]), a GROUP clause [Page 199], or HAVING clause [Page 200] must **not** be specified.
- Expressions must not contain a set function (set_function_spec) [Page 105].
- See also the section entitled "Updateable result table" under QUERY statement [Page 184].

ORDER clause

The ORDER clause [Page 202] specifies a sort sequence for a result table.

UPDATE clause

An UPDATE clause [Page 203] can only be specified for updateable result tables. For updateable result tables, a position within a particular result table always corresponds to a position in the underlying tables and thus, ultimately, to a position in one or more base tables.

If an UPDATE clause was specified, the base tables can be updated using the position in the result table (`CURRENT OF <result table name>`) by means of an UPDATE statement [Page 178] or a DELETE statement [Page 181]. A lock can be requested for the affected lines of each of the affected base tables using a LOCK statement [Page 215].

LOCK option

The LOCK option [Page 203] determines which locks are to be set on the read rows.

FOR REUSE

If the result table is to be specified in the FROM clause of a subsequent QUERY statement [Page 184], the table should be specified with FOR REUSE keywords. If FOR REUSE is not specified, the reusability of the result table depends on internal system strategies.

Since specifying FOR REUSE increases the response times of some QUERY statements, it should only be specified if it is required to reuse the result table.

See also:

SELECT statement (select_statement) [Page 188]

SELECT Statement (select_statement)

A SELECT statement (`select_statement`) defines and creates an unnamed result table (see named/unnamed result table [Page 185]).

Syntax

```
<select_statement> ::= <query_expression> [<order_clause>]
[<update_clause>] [<lock_option>] [FOR REUSE]
```

query expression [Page 189], order clause [Page 202], update clause [Page 203], lock option [Page 203]

Explanation

An OPEN CURSOR statement [Page 204] is not permitted for result tables created with this SELECT statement.

The SELECT statement (`select_statement`) is subject to the rules that were specified for the DECLARE CURSOR statement [Page 185] and those that were specified for the OPEN CURSOR statement.

Depending on the search strategy, either all the rows in the result table are searched when the SELECT statement (`select_statement`) is executed and the result table is physically generated, or each next result table row is searched when a FETCH statement [Page 205] is executed, without being physically stored. This must be taken into account for the time behavior of FETCH statements.

Updateable result table

A result table or the underlying base tables are updateable if the QUERY statement satisfies the following conditions:

- The QUERY statement [Page 184] comprises a DECLARE CURSOR statement.
- The QUERY expression (`query_expression`) must only comprise one QUERY specification (query_spec) [Page 192].
- Only one base table or one updateable view table may be specified in the FROM clause [Page 196] of the QUERY specification (`query_spec`).
- The DISTINCT keyword (see DISTINCT specification [Page 193]), a GROUP clause [Page 199], or HAVING clause [Page 200] must **not** be specified.
- Expressions must not contain a set function (set_function_spec) [Page 105].
- See also the section entitled "Updateable result table" under QUERY statement [Page 184].

ORDER clause

The ORDER clause [Page 202] specifies a sort sequence for a result table.

UPDATE clause

An UPDATE clause [Page 203] can only be specified for updateable result tables. For updateable result tables, a position within a particular result table always corresponds to a position in the underlying tables and thus, ultimately, to a position in one or more base tables.

If an UPDATE clause was specified, the base tables can be updated using the position in the result table (`CURRENT OF <result table name>`) by means of an UPDATE statement [Page 178] or a DELETE statement [Page 181]. A lock can be requested for the affected lines of each of the affected base tables using a LOCK statement [Page 215].

LOCK option

The LOCK option [Page 203] determines which locks are to be set on the read rows.

FOR REUSE

If the result table is to be specified in the FROM clause of a subsequent QUERY statement [Page 184], the table should be specified with FOR REUSE keywords. If FOR REUSE is not specified, the reusability of the result table depends on internal system strategies.

Since specifying FOR REUSE increases the response times of some QUERY statements, it should only be specified if it is required to reuse the result table.

See also:

SELECT Statement (named_select_statement) [Page 186]

QUERY expression (query expression)

QUERY expressions are required to generate an unordered result table in a SELECT statement [Page 188].

Syntax

```
<query_expression> ::= <query_term> | <query_expression> UNION [ALL]
<query_term> | <query_expression> EXCEPT [ALL] <query_term>
```

query_term [Page 190]

Explanation

If the QUERY expressions consists of only one QUERY specification (query spec) [Page 192] (specified in `query term`), the result of the expression is the unchanged result of the QUERY specification.

If a QUERY expression consists of more than one QUERY specification, the number of selected columns in all QUERY specifications of the QUERY expression must be the same. The respective i^{th} selected columns of the QUERY specifications must be comparable.

Column type (select column)	
Numeric columns	Are comparable. If all i^{th} selected columns are numeric columns, the i^{th} column of the result table is a numeric column.
Alphanumerical column, code attribute [Page 17] BYTE	Are comparable.
Alphanumerical column, code attribute ASCII, EBCDIC, UNICODE	Are comparable. Are also comparable with date, time, and timestamp values.
All i^{th} columns are date values [Page 17]	The i^{th} column of the result table is a date value.
All i^{th} columns are time values [Page 17]	The i^{th} column of the result table is a time value.

All i^{th} columns are timestamp values [Page 17]	The i^{th} column of the result table is a timestamp value.
Columns of the type BOOLEAN [Page 17]	Are comparable.
All i^{th} columns are of the type BOOLEAN	The i^{th} column of the result table is of the type BOOLEAN.
Columns of any other data type (not mentioned above)	The i^{th} column of the result table is an alphanumeric column. Comparable columns with differing code attributes are converted.

If columns are comparable but have different lengths, the corresponding column of the result table has the maximum length of the underlying columns.

Column names in the result table

The names of the result table columns are formed from the names of the selected columns of the first QUERY specification.

Let T1 be the left operand of UNION, EXCEPT, or INTERSECT (defined in `query term`). Let T2 be the right operand. Let R be the result of the operation on T1 and T2.

- A row is a duplicate of another row if both have identical values in each column. NULL values [Page 15] are assumed to be identical. Special NULL values [Page 15] are assumed to be identical.
- UNION: R contains all rows from T1 and T2.
- EXCEPT: R contains all rows from T1 which have no duplicate rows in T2.
- INTERSECT: R contains all rows from T1 which have a duplicate row in T2. A row from T2 can only be a duplicate row of exactly one row from T1. More than one row from T1 cannot have the same duplicate row in T2.
- DISTINCT is implicitly assumed for the QUERY expressions belonging to T1 and T2 if ALL is not specified. All duplicate rows are removed from R.

If parentheses are missing, then INTERSECT will be evaluated before UNION and EXCEPT. UNION and EXCEPT have the same precedence and will be evaluated from left to right in the case that parentheses are missing.

QUERY term (query_term)

A QUERY term (`query_term`) is part of the syntax in a QUERY expression (`query_expression` or `named_query_expression`).

Syntax

```
<query_term> ::= <query_primary> | <query_term> INTERSECT [ALL]
<query_primary>

<query_primary> ::= <query_spec> | (<query_expression>)
```

query_spec [Page 192], query_expression [Page 189]

Explanation

See QUERY expression [Page 189] or QUERY expression (named_query_expression) [Page 191]

QUERY expression (named query expression)

QUERY expressions (named_query_expressions) are required to generate an unordered result table in a SELECT statement (named_select_statement) [Page 186].

Syntax

```
<named_query_expression> ::= <named_query_term> |
<named_query_expression> UNION [ALL] <query_term> |
<named_query_expression> EXCEPT [ALL] <query_term>
```

named_query_term [Page 192], query_term [Page 190]

Explanation

If the QUERY expressions consists of more than one QUERY specification (query_spec) [Page 192] (specified in named_query_term or in query_term), the only the first QUERY specification in the QUERY expression may be a QUERY specification (named_query_spec).

If the QUERY expressions consists of only one QUERY specification (named_query_spec) [Page 195] (specified in named_query_term), the result of the expression is the unchanged result of the QUERY specification.

If a QUERY expression consists of more than one QUERY specification, the number of selected columns in all QUERY specifications of the QUERY expression must be the same. The respective i^{th} selected columns of the QUERY specifications must be comparable.

Column type (select column)	
Numeric columns	Are comparable. If all i^{th} selected columns are numeric columns, the i^{th} column of the result table is a numeric column.
Alphanumerical column, code attribute [Page 17] BYTE	Are comparable.
Alphanumeric columns, code attribute ASCII, EBCDIC, UNCODE	Are comparable. Are also comparable with date, time, and timestamp values.
All i^{th} columns are date values [Page 17]	The i^{th} column of the result table is a date value.
All i^{th} columns are time values [Page 17]	The i^{th} column of the result table is a time value.
All i^{th} columns are timestamp values [Page 17]	The i^{th} column of the result table is a timestamp value.
Columns of the type BOOLEAN [Page 17]	Are comparable.
All i^{th} columns are of the type BOOLEAN	The i^{th} column of the result table is of the type BOOLEAN.
Columns of any other data type (not mentioned above)	The i^{th} column of the result table is an alphanumeric column. Comparable columns with differing code attributes are converted.

If columns are comparable but have different lengths, the corresponding column of the result table has the maximum length of the underlying columns.

Column names in the result table

The names of the result table columns are formed from the names of the selected columns of the first QUERY specification.

Let T1 be the left operand of UNION, EXCEPT, or INTERSECT (defined in `named_query_term`). Let T2 be the right operand. Let R be the result of the operation on T1 and T2.

- A row is a duplicate of another row if both have identical values in each column. NULL values [Page 15] are assumed to be identical. Special NULL values [Page 15] are assumed to be identical.
- UNION: R contains all rows from T1 and T2.
- EXCEPT: R contains all rows from T1 which have no duplicate rows in T2.
- INTERSECT: R contains all rows from T1 which have a duplicate row in T2. A row from T2 can only be a duplicate row of exactly one row from T1. More than one row from T1 cannot have the same duplicate row in T2.
- DISTINCT is implicitly assumed for the QUERY expressions belonging to T1 and T2 if ALL is not specified. All duplicate rows are removed from R.

If parentheses are missing, then INTERSECT will be evaluated before UNION and EXCEPT. UNION and EXCEPT have the same precedence and will be evaluated from left to right in the case that parentheses are missing.

QUERY term (named query term)

A QUERY term (`named_query_term`) is part of the syntax in a QUERY expression (`named_query_expression`).

Syntax

```
<named_query_term> ::= <named_query_primary> | <named_query_term>
INTERSECT [ALL] <query_primary>

<named_query_primary> ::= <named_query_spec> |
(<named_query_expression>)
<query_primary> ::= <query_spec> | (<query_expression>)
```

named_query_spec [Page 195], named_query_expression [Page 191], query_spec [Page 192], query_expression [Page 189]

Explanation

See QUERY expression (named_query_expression) [Page 191]

QUERY specification (query_spec)

QUERY specifications (`query_spec`) are required to generate an unordered result table in a SELECT statement [Page 188]. A QUERY specification is part of the syntax in a QUERY term (query_term) [Page 190] or QUERY term (named_query_term) [Page 192].

Syntax

```
<query_spec> ::= SELECT [<distinct_spec>] <select_column>,...
<table_expression>
```

distinct_spec [Page 193], select_column [Page 193], table_expression [Page 195]

Explanation

A QUERY specification specifies a result table. The result table is generated from a temporary result table. The temporary result table is the result of the table expression [Page 195].

DISTINCT function (distinct spec)

The DISTINCT specification (`distinct_spec`) is specified in a QUERY specification (query_spec) [Page 192], QUERY specification (named_query_spec) [Page 195], or SINGLE SELECT statement [Page 208] to remove duplicate rows.

Syntax

```
<distinct_spec> ::= DISTINCT | ALL
```

Explanation

A row is a duplicate of another row if both have identical values in each column. NULL values [Page 15] are assumed to be identical. Special NULLvalues [Page 15] are also assumed to be identical.

DISTINCT: all duplicate rows are removed from the result table.

ALL: no duplicate rows are removed from the result table.

If no DISTINCT specification is specified, no duplicate rows are removed from the result table.

Selected Column (select_column)

Selected columns (`select_column`) must be specified in a QUERY specification (query_spec) [Page 192] or a QUERY specification (named_query_spec) [Page 195] to specify a result table.

The sequence of selected columns defines the columns in the result table. The columns in the result table are produced from the columns of the temporary result table and by the ROWNO columns or STAMP columns, if these exist. The columns of the temporary result table are determined by the FROM clause [Page 196] of the table expression [Page 195]. The order of the column names in the temporary result table is determined by the order of the table names in the FROM clause.

Syntax

```
<select_column> ::= <table_columns> | <derived_column> |
<rowno_column> | <stamp_column>

<table_columns> ::= * | <table_name>.* | <reference_name>.*
<derived_column> ::= <expression> [ [AS] <result_column_name>] |
<result_column_name> = <expression>
<rowno_column> ::= ROWNO [<result_column_name>] |
<result_column_name> = ROWNO
<stamp_column> ::= STAMP [<result_column_name>] |
<result_column_name> = STAMP

<result_column_name> ::= <identifier>
```

table_name [Page 47], reference_name [Page 45], expression [Page 52], identifier [Page 36]

Explanation

Every column name that is specified as a selected column must uniquely denote a column in a QUERY specification (query spec) [Page 192] of the underlying tables. If necessary, the column name must be qualified with the table name.

The specification of a column with the data type LONG [Page 121] in a selected column is only valid in the uppermost sequence of selected columns in a QUERY statement [Page 184] or SINGLE SELECT statement [Page 208] if the DISTINCT specification [Page 193] was not used there.

The specification of a column with the data type LONG in a selected column is only valid in the uppermost sequence of select columns in a CREATE VIEW statement [Page 143] which is based on exactly one base table.

If a selected column contains a set function (set function spec) [Page 105], the sequence of selected columns to which the selected column belongs must not contain any table columns, and every column name occurring in an expression [Page 52] must denote a grouping column, or the expression must consist of grouping columns.

It is possible to specify scalar subqueries [Page 201].

- Specifying table columns in a selected column is a quick way of specifying the result table columns.
  - Specifying a selected column of the type * is a quick way of specifying all temporary result table columns.
    Columns for which the user has not the SELECT privilege and the implicitly generated column SYSKEY are not passed.
  - Specifying `<table_name>.*` or `<reference_name>.*` is quick way of specifying all the columns in the underlying table. The first column name of the result table is taken from the first column name of the underlying table, the second column name of the result table corresponds to the second column name of the underlying table, etc. The order of column names in the underlying table corresponds to the order determined when the underlying table is defined. Columns for which the user has not the SELECT privilege and the implicitly generated column SYSKEY are not passed.
- Specifying **`derived column`** in a selected column defines a column in the result table. If a column of the result table has the form `<expression> [AS] <result_column_name>` or the form `<result_column_name> = <expression>`, this result column receives the name `result_column_name`.
  If no `<result_column_name>` is specified and the expression [Page 52] is a column specification that denotes a column in the temporary result table, the column in the result table receives the column name of the temporary result table.
  If no `<result_column_name>` is specified and the expression is not a column specification, the column receives the name `EXPRESSION_`, where "_" denotes a number with a maximum of three digits, starting with `EXPRESSION1`, `EXPRESSION2`, etc.
- A ROWNO column may only be used in a selected column that belongs to a QUERY statement.
  If a ROWNO column is specified, a column with the data type FIXED [Page 121](10) is generated with the name ROWNO. It contains the values 1, 2, 3,... which the numbers of the result table rows.
  If the ROWNO column was specified in the form `ROWNO <result_column_name>` or the form `<result_column_name> = ROWNO`, this result column receives the name `result_column_name`.
  A ROWNO column>must not be ordered by using ORDER BY.
- A STAMP column may only be specified in a selected column that belongs to a QUERY- expression [Page 189] of an INSERT statement [Page 173].
  The database system is able to generate unique values. This is a serial number that

starts with X'000000000001'. The values are assigned in ascending order. It cannot be ensured that a sequence of values is uninterrupted.
If a STAMP column is specified, the next value of the data type CHAR [Page 120](8) BYTE generated by the database system is produced for each row in the temporary result table.

Each column of a result table has exactly the same data type, the same length, the same precision, and the same scale as the `derived column` or the column underlying the table columns.

This does not apply to the data types DATE and TIMESTAMP. To enable the representation of any date and time format, the length of the result table column is set to the maximum length required for the representation of a date value [Page 17] (length 10) or a timestamp value [Page 17] (length 26).

QUERY specification (named_query_spec)

QUERY specifications (`named_query_spec`) are required to generate an unordered result table in a SELECT statement (named_select_statement) [Page 186]. A QUERY specification is part of the syntax in a QUERY term (named_query_term) [Page 192].

Syntax

```
<named_query_spec> ::= SELECT [<distinct_spec>] <result_table_name>
(<select_column>,...) <table_expression>
```

distinct_spec [Page 193], result_table_name [Page 41], select_column [Page 193], table_expression [Page 195]

Explanation

A QUERY specification specifies a result table with the name `result_table_name`. The result table is generated from a temporary result table. The temporary result table is the result of the table expression [Page 195].

Table expression

A table expression specifies a single or a simple or grouped result table (see result table name [Page 41]).

Syntax

```
<table_expression> ::= <from_clause> [<where_clause>]
[<group_clause>] [<having_clause>]
```

from_clause [Page 196], where_clause [Page 198], group_clause [Page 199], having_clause [Page 200]

Explanation

A table expression produces a temporary result table. If there are no optional clauses, this temporary result table is the result of the FROM clause. Otherwise, each specified clause is applied to the result of the previous condition and the table is the result of the last specified clause. The temporary result table contains all of the columns in all the tables listed in the FROM clause.

The order of the GROUP and HAVING clauses is random.

FROM clause

A FROM clause specifies a table in a table expression [Page 195] that is formed from one or more tables.

Syntax

```
<from_clause> ::= FROM <from_table_spec>,...
```

from_table_spec [Page 196]

Explanation

The FROM clause specifies a table. This table can be derived from several base, view, and result tables (see Table [Page 23]). The number of underlying tables in a FROM clause is equal to the total number of underlying tables in each FROM TABLE specification. The number of underlying tables in a FROM clause must not exceed 64.

The user must have the SELECT privilege for each specified table or for at least one column in the specified table.

The result of a FROM clause is a table that is generated from the specified tables as follows:

- If the FROM clause comprises a single FROM TABLE specification, the result is the specified table.
- If the FROM clause contains more than one FROM TABLE specification, a result table is built that includes all possible combinations of all rows of the first table with all rows of the second table, etc. From a mathematical perspective, this is the Cartesian product of all the tables.

This rule describes the effect of the FROM clause, not its actual implementation.

FROM TABLE specification (from_table_spec)

Each FROM TABLE specification (`from_table_spec`) in a FROM clause [Page 196] specifies no, one, or any number of table identifiers.

Syntax

```
<from_table_spec> ::= <table_name> [<reference_name>]
| <result_table_name> [<reference_name>]
| (<query_expression>) [<reference_name>]
| <joined_table>
```

table_name [Page 47], reference_name [Page 45], result_table_name [Page 41], query_expression [Page 189], joined_table [Page 197]

Explanation

Reference name

If a FROM TABLE specification does not contain a reference name, the table name or result table name is the table identifier.

If a FROM TABLE specification contains a reference name, the reference name is the table identifier.

Each reference name must be different from each identifier [Page 36] that specifies a table name. If a result table name is a table identifier, there must not be any table identifiers in the form `<table_name>` equal to `[<owner.]<result_table_name>`, where owner [Page 41] is the current user. Each table identifier must differ from any other table identifier.

The validity range of the table identifiers is the entire QUERY specification [Page 192] within which the FROM TABLE specification is used. If column names are to be qualified within the QUERY specification, table identifiers must be used for this purpose.

Reference names are essential for formulating JOIN conditions within a table. For example, `FROM HOTEL, HOTEL X` defines a reference name X for the second occurrence of the HOTEL table. Reference names are also necessary sometimes to formulate correlated subqueries [Page 200]. Similarly, a reference name is required if a column in the result of a QUERY expression can only be identified uniquely by specifying the reference name.

Number of underlying tables

If a FROM TABLE specification denotes a base table, result table, or the result of a QUERY expression, the number of tables underlying this FROM TABLE specification is equal to 1.

If a FROM TABLE specification denotes a complex view table, the number of tables underlying this FROM TABLE specification is equal to 1.

If a FROM TABLE specification denotes a view table that is not a complex view table, the number of underlying tables is equal to the number of tables underlying the FROM condition [Page 196] of the view table.

If a FROM TABLE specification denotes a JOINED TABLE, the number of tables underlying this FROM TABLE specification is equal to the total number of underlying tables of the FROM TABLE specifications contained in it.

QUERY expression (query_expression)

A FROM TABLE specification that contains a QUERY expression specifies a table identifier only if a reference name is specified.

If a FROM TABLE specification contains a QUERY expression, a result table is built that matches this QUERY expression. This result table obtains a system-internal name that collides neither with an unnamed nor a named result table. While the FROM condition is being processed, the result of the QUERY expression is used in the same way as a named result table and is deleted implicitly after processing.

A table expression [Page 195] containing at least one OUTER JOIN indicator (see JOIN predicate [Page 62]) or OUTER JOIN TYPE (LEFT | RIGHT | FULL) (see joined_table [Page 197]) is subject to strict restrictions if it is to be based on more than two tables. For this reason, a QUERY expression is frequently required to formulate a QUERY specification [Page 192] that is to be based on at least three tables and in which at least one OUTER JOIN indicator is used in a JOIN predicate.

JOINED TABLE

A FROM TABLE specification containing a JOINED TABLE (joined_table [Page 197]) specifies the number of table identifiers that are specified by the FROM TABLE specifications it contains.

joined_table

JOINED TABLE (`joined table`) can be specified as part of a FROM TABLE specification (from table spec) [Page 196].

Syntax

```
<joined_table> ::=
<from_table_spec> CROSS JOIN <from_table_spec>
| <from_table_spec> [INNER] JOIN <from_table_spec> <join_spec>
| <from_table_spec> [<LEFT|RIGHT|FULL> [OUTER]] JOIN
<from_table_spec> <join_spec>

<join_spec> ::= ON <search_condition>
```

from_table_spec [Page 196], search_condition [Page 70], column_name [Page 46]

Explanation

If a FROM TABLE specification comprises a JOINED TABLE, the result is generated as follows:

Let FT1 be the set of all rows in the table specified by the first FROM TABLE specification.
Let FT2 be the set of all rows in the table specified by the second FROM TABLE specification.

- If JOINED TABLE is specified as **CROSS JOIN**, a table is created that comprises all possible combinations of FT1 and FT2. From a mathematical perspective, the Cartesian product of the two tables is calculated.
- If JOINED TABLE is specified with the keyword JOIN without the optional keywords INNER, LEFT, RIGHT, FULL, or OUTER, the JOIN type is assumed to be INNER.

Let T be the set of result rows consisting of all possible combinations of FT1 and FT2. Each result row satisfies the JOIN specification for this set.

- If JOINED TABLE is specified with the JOIN type **INNER**, the result is the set T.
- If JOINED TABLE is specified with the JOIN type **LEFT**, the result is the set T plus the rows from FT1 that are not in T. The result columns that are not formed from FT1 are assigned the NULL value.
- If JOINED TABLE is specified with the JOIN type **RIGHT**, the result is the set T plus the rows from FT2 that are not in T. The result columns that are not formed from FT2 are assigned the NULL value.
- If JOINED TABLE is specified with the JOIN type **FULL**, the result is the set T plus the rows from FT1 that are added by the JOIN types LEFT and RIGHT.

The rules specified for the WHERE condition [Page 198] apply to the JOIN specification (`join_spec`) `ON <search_condition>` with the restriction that no links using the Boolean operator OR are permitted.

WHERE Clause (where_clause)

The WHERE clause specifies the conditions for building a result table (see result table name [Page 41]).

Syntax

```
<where_clause> ::= WHERE <search_condition>
```

search_condition [Page 70]

Explanation

The search condition is applied to each row in the temporary result table formed by the FROM clause [Page 196]. The result of the WHERE clause is a table that only contains those rows from the result table for which the search condition is true.

The search condition may only contain column specifications for which the user has the SELECT privilege.

Each column specification [Page 47] directly contained in the search condition must uniquely identify a column from the tables specified in the FROM clause of the table expression [Page 195]. If necessary, the column name must be qualified with the table identifier. If reference names are defined in the FROM clause for table names, they must be used as table identifiers in the search condition.

Expressions [Page 52] in the search condition must not contain a set function (set_function_spec) [Page 105], except in the exception below.

```
SELECT ... FROM uppertab,…
HAVING ... (SELECT ...
                    WHERE MIN(uppertab, ...)...
           )
```

The SELECT statement is allowed in the specified format.

HAVING clause [Page 200]: In a subquery [Page 200] used in this clause, it is possible to use WHERE clauses that contain Set functions for the columns of the table used in the SELECT … HAVING statement (Correlated Subquery [Page 200]).

In the case of correlated subqueries, a column specification can identify a column in a table that was specified in a FROM clause of a different table expression in the QUERY specification [Page 192].

Each subquery [Page 200] in the search condition is usually evaluated only once. In the case of a correlated subquery, the subquery is executed for each row in the result table generated by the FROM clause.

GROUP Clause (group_clause)

The GROUP clause specifies grouping criteria for a result table (see result table name [Page 41]).

Syntax

```
<group_clause> ::= GROUP BY <expression>,...
```

expression [Page 52]

Explanation

Each column name specified in the GROUP clause must identify a `result_column_name` in the selected columns [Page 193] of the QUERY specification [Page 192] or uniquely identify a column in the tables on which the QUERY specification is based. If necessary, the column name must be qualified with the table identifier.

The GROUP clause allows the functions SUM [Page 109], AVG [Page 108], MAX/MIN [Page 109], COUNT [Page 108], STDDEV [Page 109], and VARIANCE [Page 109] to be applied not only to entire result tables but also to groups of rows within a result table. A group is defined by the grouping columns specified in GROUP BY. All rows of a group have the same values in the grouping columns. Rows containing the NULL value [Page 15] in a grouping column are combined to form a group. The same is true for the special NULL value [Page 15].

GROUP BY generates one row for each group in the result table. The selected columns in the QUERY specification, therefore, may only contain those grouping columns and operations on grouping columns, as well as those expressions [Page 52] that use the functions SUM, AVG, MAX/MIN, COUNT, STDDEV, and VARIANCE.

If no rows satisfy the conditions indicated in the WHERE clause [Page 198] and a GROUP clause was specified, the result table is empty.

Specifying scalable subqueries [Page 201] is not permissible in a GROUP clause.

See also:

Restrictions [Page 219]

HAVING clause

The HAVING clause specifies the properties of a group.

Syntax

```
<having_clause> ::= HAVING <search_condition>
```

search_condition [Page 70]

Explanation

Each expression [Page 52] in the search condition that does not occur in the argument of a set function (set_function_spec) [Page 105] must identify a grouping column.

If the HAVING clause is used without a GROUP clause [Page 199], the result table built so far is regarded as a group.

The search condition is applied to each group in the result table. The result of the HAVING clause is a table that only contains those groups for which the search condition is true.

Subquery

A subquery specifies a result table (see result table name [Page 41]) that can be used in certain predicates and for updating column values.

Syntax

```
<subquery> ::= (<query_expression>)
```

query_expression [Page 189]

Explanation

Subqueries can be used in a SET UPDATE condition [Page 180] of an UPDATE statement [Page 178]. In this case, the subquery must produce a result table that contains a maximum of one row.

Subqueries can be used in an INSERT statement (INSERT statement) [Page 173]).

Subqueries can be used in the following predicates:

Comparison predicate [Page 58]
EXISTS predicate [Page 60]
IN predicate [Page 61]
Quantified predicate [Page 67]

See also:

Scalar Subquery [Page 201]

Correlated Subquery [Page 200]

Correlated Subquery

Certain predicates can contain subqueries. These subqueries, in turn, can contain other subqueries, etc. A subquery [Page 200] with further subqueries is the higher-level subquery of the subqueries it contains.

- The search condition [Page 70] of a subquery can contain column names that belong to tables that are contained in higher levels in the FROM clause [Page 196]. This type of subquery is called a **correlated subquery**.

- Tables that are used in subqueries in this way are called **correlated tables**. No more than 16 correlated tables are allowed within an SQL statement.
- Columns that are used in subqueries in this way are called **correlated columns**. A total of 64 correlated columns can be used in an SQL statement.

If the qualifying table name or reference name does not clearly identify a table at a higher level, the table at the lowest level is taken from these non-unique tables.

If the column name is not qualified by the table name or reference name, the tables at higher levels are searched. The column name must be unique in all tables of the FROM clause to which the table found belongs.

If a correlated subquery is used, the values of one or more columns in a temporary result row at a higher level are included in the search condition of a subquery at a lower level, whereby the result of the subquery is used to uniquely qualify the higher-level temporary result row.

Example tables hotel [Page 112] and room [Page 113].
For every city, the names of all hotels are searched which have single room prices less than the average price of the city concerned.

```
SELECT name, city FROM hotel X, room
WHERE X.hno = room.hno AND room.roomtype = 'SINGLE' AND
room.price <
(SELECT AVG(room.price) FROM hotel, room
WHERE hotel.hno = room.hno AND hotel.city = X.city AND
room.roomtype = 'SINGLE' )
```

Name	City
Eight Avenue	Los Angeles
Sunshine	Los Angeles
Atlantic	New York

Scalar Subquery (scalar_subquery)

A scalar subquery (`scalar_subquery`) is a special subquery [Page 200].

Syntax

```
<scalar_subquery> ::= <subquery>
```

subquery [Page 200]

Explanation

Scalar subqueries are produced through the restriction of the result set of a result table [Page 41] to a maximum of one value.

Scalar subqueries can be used as expressions (see factor [Page 54]).

Scalar subqueries are not allowed in a GROUP clause [Page 199] or an ORDER clause [Page 202].

Scalar subquery in the list of the values to be inserted in an INSERT statement for the table hotel [Page 112]:

```
INSERT hotel VALUES((SELECT MAX(hno)+10 FROM hotel), 'Three
Seasons', 90014, 'Los Angeles', '247 Broad Street')
```

Scalar subquery in a selected column [Page 193]:

```
SELECT hno, price, (SELECT MIN(price) FROM room) FROM room
```

ORDER Clause (order_clause)

The ORDER clause specifies a sort sequence for a result table (see result table name [Page 41]).

Syntax

```
<order_clause> ::= ORDER BY <sort_spec>,...
```

```
<sort_spec> ::= <unsigned_integer> [ASC | DESC] | <expression> [ASC |
DESC]
```

unsigned_integer [Page 34], expression [Page 52]

Explanation

The sort columns specified in the ORDER clause determine the sequence of the sort criteria.

A number n specified in the sorting specification (`sort_spec`) identifies the n^{th} column in the result table. n must be less than or equal to the number of columns in the result table.

Scalable subqueries [Page 201] are not permissible in an ORDER clause.

ASC/DESC

ASC: the values are sorted in ascending order.

DESC: the values are sorted in descending order.

The default setting is ASC.

Further information

If a QUERY expression [Page 189] consists of more than one QUERY specification [Page 192], sort specifications must be specified in the form `<unsigned_integer> [ASC | DESC]`.

If a QUERY specification was specified with DISTINCT, the total of the internal lengths of all sorting columns must not exceed 1016 characters; otherwise it can comprise 1020 characters.

Column names in the sort specifications must be columns in the tables of the FROM clause [Page 196] or a `result_column_name` in the selected columns [Page 193] of the QUERY specification.

If DISTINCT or a set function [Page 105] in a selected column was used, the sort specification must identify a column in the result table.

Values are compared in accordance with the rules for the comparison predicate [Page 58]. For sorting purposes, NULL value [Page 15]s are greater than non-NULL values, and special NULL value [Page 15]s are greater than non-NULL values but less than NULL values.

See also:

Restrictions [Page 219]

UPDATE Clause (update_clause)

The UPDATE clause specifies that a result table (see result table name [Page 41]) is to be updateable. The system sets exclusive locks for all rows that form this result table.

Syntax

```
<update_clause> ::= FOR UPDATE [OF <column_name>,...] [NOWAIT]
```

column_name [Page 46]

Explanation

The specified column names must identify columns in the tables underlying the QUERY specification [Page 192]. They do not have to occur in a selected column [Page 193].

The QUERY statement that contains the UPDATE clause must generate an updateable result table.

The UPDATE clause is a prerequisite for using the results table with `CURRENT OF <result_table_name>` in the UPDATE statement [Page 178], in the DELETE statement [Page 181], and in the LOCK statement [Page 215]. The UPDATE clause is meaningless for other forms of the above mentioned SQL statements, as well as in interactive mode.

All columns of the underlying base tables are updateable if the user has the corresponding privileges, irrespective of whether they were specified as a column name [Page 46].

For performance reasons, it is recommended to specify column names only if the cursor is to be used in an UPDATE statement.

Assume that the a column x fulfills the following conditions:

- x is contained in the primary key or an index
- x is contained in the search condition [Page 70] of the QUERY statement
- x is contained in a SET UPDATE clause [Page 180] of the UPDATE statement in the type `x = <expression>`, where the expression [Page 52] contains the column x.

If all of the conditions are fulfilled, it is essential that you specify the column x as a column name in the UPDATE clause.

If at least one of these conditions is not satisfied, the column name should not be specified.

NOWAIT

If `NOWAIT` is not specified and a lock collision occurs, the system waits for the locked data object to be released (but only as long as is specified by the database parameter REQUEST_TIMEOUT).

If `NOWAIT` is specified, the database system does not wait until another user has released a data object. Instead, it returns a message if a collision occurs. If there is no collision, the requested lock is set.

LOCK Option (lock_option)

The LOCK option requests a lock for each selected row.

Syntax

```
<lock_option> ::=
  WITH LOCK [(IGNORE)|(NOWAIT)] [EXCLUSIVE|OPTIMISTIC] [ISOLATION
LEVEL <unsigned_integer>]
```

unsigned_integer [Page 34] may only have the values 0, 1, 2, 3, 10, 15, 20, or 30

Explanation

IGNORE

If `(IGNORE)` is not specified and a lock collision occurs, the system waits for a locked row to be released (but only as long as is specified by the database parameter REQUEST_TIMEOUT).

If `(IGNORE)` is specified, the system does not wait for a locked row to be released by another transaction. Instead, it ignores this row if a lock collision occurs. If there is no collision, the requested lock is set. `(IGNORE)` can only be specified in isolation level 1.

NOWAIT

If `(NOWAIT)` is not specified and a lock collision occurs, the system waits for the locked data object to be released (but only as long as is specified by the database parameter REQUEST_TIMEOUT).

If `(NOWAIT)` is specified, the database system does not wait until another user has released a data object. Instead, it returns a message if a collision occurs. If there is no collision, the requested lock is set.

EXCLUSIVE

An exclusive lock is defined. As long as the locked row has not been changed or deleted, the exclusive lock can be released using the UNLOCK statement [Page 216].

OPTIMISTIC

An optimistic lock is defined on rows. This is only meaningful in connection with isolation levels 0, 1, 10, and 15.

Share Lock

If neither EXCLUSIVE nor OPTIMISTIC is specified, a share lock is set on the corresponding rows.

ISOLATION LEVEL

The locks are set independently of the ISOLATION specification (`isolation_spec`) of the CONNECT statement [Page 211]. The isolation level of the LOCK option can have a higher or lower value than that in the CONNECT statement.

If an isolation level is specified by the LOCK option, it is only valid for the duration of the SQL statement which contains the LOCK option specification. Afterwards, the isolation level that was specified in the CONNECT statement is applicable again. In the case of a SELECT statement (named_select_statement) [Page 186], SELECT statement (select_statement) [Page 188], or an OPEN CURSOR statement [Page 204], for which the result table is not actually physically generated, the specified isolation level is valid for this SQL statement and all FETCH statements [Page 205] that refer to the result table. The isolation level that was specified in the CONNECT statement is applicable for other SQL statements that were executed in the meantime.

See also:

Lock Behavior

OPEN CURSOR statement

An OPEN CURSOR statement generates the result table defined under the specified name with a DECLARE CURSOR statement [Page 185].

Syntax

```
<open_cursor_statement> ::= OPEN <result_table_name>
```

result_table_name [Page 41]

Explanation

Existing result tables are implicitly deleted when a result table is generated with the same name.

All result tables generated within the current transaction are implicitly deleted at the end of the transaction using the ROLLBACK statement [Page 213].

All result tables are implicitly deleted at the end of the session using the RELEASE statement [Page 217]. A CLOSE statement [Page 208] can be used to delete them explicitly beforehand.

If the name of a result table is identical with that of a base table, view table (see Table [Page 23]), or a synonym [Page 24], these tables cannot be accessed as long as the result table exists.

At any given time when a result table is processed, there is a position which may be before the first row, on a row, after the last row or between two rows. After generating the result table, this position is before the first row of the result table.

Depending on the search strategy, either all the rows in the result table are searched when the OPEN CURSOR statement is executed and the result table is physically generated, or each next result table row is searched when a FETCH statement [Page 205] is executed, without being physically stored. This must be considered for the time behavior of OPEN CURSOR statements and FETCH statements.

If the result table is empty, the return code `100 - row not found` - is set.

The number of rows in the result table is returned in the SQLCA in the third entry of SQLERRD. If this counter has the value -1, there is at least one result row.

FETCH statement

The FETCH statement assigns the values of the current row in a result table (see result table name [Page 41]) to parameters.

Syntax

```
<fetch_statement> ::=
FETCH [FIRST | LAST | NEXT | PREV | <position> | SAME]
[<result_table_name>] INTO <parameter_spec>,...

<position> ::= POS (<unsigned_integer>) | POS (<parameter_spec>)
| ABSOLUTE <integer> | ABSOLUTE <parameter_spec>
| RELATIVE <integer> | RELATIVE <parameter_spec>
```

result_table_name [Page 41], parameter_spec [Page 48], unsigned_integer [Page 34], integer [Page 34]

Explanation

If no result table name is specified, the FETCH statement refers to the last unnamed result table that was generated (see named/unnamed result table [Page 185]).

Depending on the search method, either all the rows in the result table are searched when the OPEN CURSOR statement [Page 204], the SELECT statement (select_statement) [Page 188], or the SELECT statement (named_select_statement) [Page 186] is executed and the result table is generated, or each subsequent row in the result table is searched when a FETCH statement is executed, but they are not physically stored. This must be taken into account for the time behavior of FETCH statements. Depending on the isolation level

selected, this can also cause locking problems with a FETCH, e.g. return code `500 - LOCK REQUEST TIMEOUT`.

Row not found

Let C be the position in the result table. The return code `100 - ROW NOT FOUND` – is output and no values are assigned to the parameters if any of the following conditions is satisfied:

- The result table is empty.
- C is positioned on or after the last result table row, and FETCH or FETCH NEXT is specified.
- C is positioned on or before the first row of the result table and FETCH PREV is specified.
- FETCH is specified with a position which does not lie within the result table.

FIRST | LAST | NEXT | PREV

- FETCH FIRST or FETCH LAST: the result table is not empty. C is positioned in the first or last row of the result table and the values of this row are assigned to the parameters.
- FETCH or FETCH NEXT: C is positioned before a row in the result table. C is positioned in this row and the values of this row are assigned to the parameters.
- FETCH or FETCH NEXT: C is positioned in a row that is not the last row in the result table. C is positioned in the next row and the values in this row are assigned to the parameters.
- FETCH PREV: C is positioned behind a row in the result table. C is positioned in this row and the values of this row are assigned to the parameters.
- FETCH PREV: C is positioned in a row that is not the first row in the result table. C is positioned in the previous row and the values in this row are assigned to the parameters.

Position: POS

Regardless of an ORDER clause [Page 202], there is an implicit order of the rows in a result table. This can be displayed by specifying a ROWNO column as a selected column [Page 193]. The specified position refers to this internal numbering.

If a position is defined with POS, the parameter specification must denote a positive integer.

If a position that is less than or equal to the number of rows in the result table was defined with POS, C is set to the corresponding row and the values of this row are assigned to the parameters. If a position that is greater than the number of rows in the result table was specified, the message `100 - ROW NOT FOUND` is output.

Position: ABSOLUTE

Let x be the value of the integer or parameter specification specified with the position. Let `abs_x` be the absolute value of x.

- FETCH ABSOLUTE and x is : FETCH ABSOLUTE is the same as a FETCH POS.
- FETCH ABSOLUTE and x=0: the return code `100 - row not found` is set.
- FETCH ABSOLUTE and x is negative: C is set after the last row of the result table where FETCH PREV is executed `abs_x` times. The last row found is the result of the SQL statement. This description refers to the logic and not the flow of the statement. If `abs_x` is larger than the number of rows in the result table, the `message 100 - ROW NOT FOUND` is output.

Position: RELATIVE

Let x be the value of the integer or parameter specification specified with the position. Let `abs_x` be the absolute value of x.

- FETCH RELATIVE and x is positive: FETCH NEXT is executed x times from the current position in the result table C.
- FETCH RELATIVE and x=0: corresponds to a FETCH SAME.
- FETCH RELATIVE and x is negative: FETCH PREV is executed `abs_x` times starting from C. This description refers to the logic and not the flow of the statement. The return code `100 - row not found` is output if one of the conditions in the section "row not found" is fulfilled.

FETCH SAME

The last row found in the result table is output again.

Parameter specification

The parameter specification [Page 48] specified in **position** must denote an integer.

The remaining parameters in the parameter specification are output parameters. The parameter identified by the n^{th} parameter specification corresponds to the n^{th} value in the current result table row. If the number of columns in this row exceeds the number of specified parameters, the column values for which no corresponding parameters exist are ignored. If the number of columns in the row is less than the number of specified parameters, no values are assigned to the remaining parameters. You must specify an indicator name [Page 42] in order to assign NULL values [Page 15] or special NULL values [Page 15].

Numbers are converted and character strings are truncated or lengthened, if necessary, to suit the corresponding parameters. If an error occurs when assigning a value to a parameter, the value is not assigned and no further values are assigned to the corresponding parameters for this FETCH statement. Any values that have already been assigned to parameters remain unchanged.

Let p be a parameter and v the corresponding value in the current row of the result table.

- v is a number: p must be a numeric parameter and v must be within the permissible range of p.
- v is a character string: p must be an alphanumeric parameter.

Further information

If no FOR REUSE was specified in the QUERY statement (see SELECT statement [Page 188]), subsequent INSERT, UPDATE, or DELETE statements that refer to the underlying base table and are executed by the current user or by other users can cause several executions of a FETCH statement to denote different rows in the result table, even though the same position was specified.

You can prevent other users from making changes by executing a LOCK statement [Page 215] for the entire table or by using the isolation level 2, 3, 15, 20, or 30 with the CONNECT statement [Page 211] or the LOCK option [Page 203] of the QUERY statement.
FOR REUSE must be specified if this is not possible or if the user makes changes to this table. Changes made in the meantime are not visible in this case.

If a result table that was physically created contains LONG columns [Page 16] and if the isolation levels 0, 1, and 15 are used, consistency between the content of the LONG columns and that of the other columns is not ensured. If the result table was not physically generated, consistency is not ensured at isolation level 0 only. For this reason, it is advisable to ensure consistency by using a LOCK statement or the isolation levels 2, 3, 20, or 30.

CLOSE statement

The CLOSE statement deletes a result table (see result table name [Page 41]).

Syntax

```
<close_statement> ::= CLOSE [<result_table_name>]
```

result_table_name [Page 41]

Explanation

- If the name of a result table is specified, this result table is deleted. This name can be used to denote another result table.
- If no result table name is specified, an existing unnamed result table is deleted.

An unnamed result table is implicitly deleted by the next SELECT statement [Page 188].

Result tables are implicitly deleted when a result table with the same name is generated.

All result tables generated within the current transaction are implicitly deleted at the end of the transaction using the ROLLBACK statement [Page 213].

All result tables are implicitly deleted at the end of the session using the RELEASE statement [Page 217].

SINGLE SELECT statement

A SINGLE SELECT statement specifies a result table with one row (see result table name [Page 41]) and assigns the values in this row to parameters.

Syntax

```
<single_select_statement> ::=
SELECT [<distinct_spec>] <select_column>,...
INTO <parameter_spec>,... FROM <from_table_spec>,...
[<where_clause>] [<group_clause>] [<having_clause>] [<lock_option>]
```

distinct_spec [Page 193], select_column [Page 193], parameter_spec [Page 48], from_table_spec [Page 196], where_clause [Page 198], group_clause [Page 199], having_clause [Page 200], lock_option [Page 203]

Explanation

The number of rows in the result table must not be greater than one. If the result table is empty or contains more than one row, corresponding messages or error codes are issued and no values are assigned to the parameters specified in the parameter specifications. The return code `100 - row not found -` is set if the result table is empty.

If the result table contains just one row, the values of this row are assigned to the corresponding parameters. The FETCH statement [Page 205] rules apply for assigning the values to the parameters.

The order of the GROUP and HAVING clauses is random.

A LONG column [Page 16] can only be specified in a selected column in the uppermost sequence of selected columns in a SINGLE SELECT statement if the DISTINCT specification [Page 193] DISTINCT was not used there.

EXPLAIN Statement (explain_statement)

The EXPLAIN statement describes the search strategy used internally by the database system for a QUERY statement [Page 184] or SINGLE SELECT statement [Page 208] (statements for searching for certain rows in specific tables). This statement indicates in particular whether and in which form key columns or indexes are used for the search.

Syntax

```
<explain statement> ::=
  EXPLAIN [(<result table name>)] <query statement>
| EXPLAIN [(<result table name>)] <single select statement>
```

result_table_name [Page 41], query_statement [Page 184], single_select_statement [Page 208]

Explanation

The EXPLAIN statement can be used to check the effect of creating or deleting indexes (see index [Page 24]) on the choice of search strategy for the specified SQL statement. It is also possible to estimate the time needed by the database system to process the specified SQL statement. The specified QUERY or SINGLE SELECT statement is not executed while the EXPLAIN statement is being executed.

You will find information on the Optimizer functions in Optimizer: SAP DB 7.4.

As a result of the EXPLAIN statement, a result table is generated (see Result table name [Page 41]). This result table may be named. If the optional name specification is missing, the result table is given the name SHOW.

Structure of the EXPLAIN result table

OWNER	CHAR [Page 120](64)
TABLENAME	CHAR(64)
COLUMN_OR_INDEX	CHAR(64)
STRATEGY	CHAR(40)
PAGECOUNT	CHAR(10)
O	CHAR (1)
D	CHAR (1)
T	CHAR (1)
M	CHAR (1)

The sequence in which the SELECT is processed is described by the order of the rows in the result table.

STRATEGY

The STRATEGY column shows which search strategy/ies is/are used and whether a result table is generated. A result table is physically generated if the column STRATEGY contains `RESULT IS COPIED` in the last result row.

COLUMN_OR_INDEX

The COLUMN_OR_INDEX column shows which key column or indexed column or which index is used for the strategy.

PAGECOUNT

The PAGECOUNT column shows which sizes are assumed for the tables or, in the case of certain strategies, for the indexes. These sizes influence the choice of the search strategy.

The assumed sizes are updated using the UPDATE STATISTICS statement [Page 217] and can be requested by selecting the system table OPTIMIZERSTATISTICS. The current sizes of tables or indexes can be checked by selecting the TABLESTATISTICS and INDEXSTATISTICS system tables. If there are large discrepancies between the values contained in the OPTIMIZERSTATISTICS and TABLESTATISTICS, the UPDATE STATISTICS statement should be performed for this table.

If the system discovers during a search in a table that the values determined by the last UPDATE STATISTICS statement are extremely low, a row is entered in the SYSUPDSTATWANTED system table that contains the table name. In all other cases, rows are entered in this system table that describe columns in tables. The UPDATE STATISTICS statement should be executed for tables and columns in tables that are described in the SYSUPDSTATWANTED system table.

The last row contains the estimated SELECT cost value in the PAGECOUNT column. The specifications for COSTLIMIT and COSTWARNING in the CREATE USER, CREATE USERGROUP, ALTER USER, and ALTER USERGROUP statements refer to this estimated SELECT cost value.

O, D, T, M

The columns O, D, T, and M are only for Support purposes. For more information, see the *Optimizer: SAP DB 7.4* documentation.

Transaction

A transaction is a sequence of SQL statements that are handled by the database system as a basic unit, in the sense that any modifications made to the database by the SQL statements are either all reflected in the state of the database, or else none of the database modifications are retained.

The first transaction is opened when a database session [Page 27] is opened with the CONNECT statement [Page 211]. The transaction is concluded with the COMMIT statement [Page 213] or the ROLLBACK statement [Page 213]. When a transaction is successfully concluded with a COMMIT statement, all of the changes to the database are retained. If a transaction is aborted using a ROLLBACK statement, on the other hand, or if it is aborted in another way, all of the changes to the database made by the transaction are rolled back.

Both the COMMIT and ROLLBACK statements open a new transaction implicitly.

A transaction can be divided into other basic units, subtransactions [Page 27].

Locks

Since the database system permits concurrent transactions on the same database objects, locks on rows, tables, and the database catalog are necessary to isolate individual transactions.

For information about the lock concept, see the *User Manual: SAP DB*, Lock Behavior section.

- The assignment of implicit locks can be affected by the setting of the isolation level with the CONNECT statement [Page 211].
- Locks can be assigned explicitly using the LOCK statement [Page 215] or by the assignment of a LOCK option [Page 203].
- Exclusive locks for rows that have not yet been modified, and share locks on rows can be released by the UNLOCK statement [Page 216] before the end of the transaction.

The locks assigned to a transaction are usually released at the end of the transaction, making the respective database objects accessible again to other transactions.

SQL statements for transaction management

CONNECT statement [Page 211]	SET statement [Page 212]	
COMMIT statement [Page 213]	ROLLBACK statement [Page 213]	SUBTRANS statement [Page 214]
LOCK statement [Page 215]	UNLOCK statement [Page 216]	RELEASE statement [Page 217]

CONNECT Statement (connect_statement)

A CONNECT statement opens a database session [Page 27] and a transaction [Page 210] for a database user.

Syntax

```
<connect_statement> ::=
  CONNECT <parameter_name> IDENTIFIED BY <parameter_name>
[<connect_option>...]
| CONNECT <parameter_name> IDENTIFIED BY <password>
[<connect_option>...]
| CONNECT <user_name> IDENTIFIED BY <parameter_name>
[<connect_option>...]
| CONNECT <user_name> IDENTIFIED BY <password> [<connect_option>...]

<connect_option> ::=
  SQLMODE <INTERNAL|ANSI|DB2|ORACLE>
| ISOLATION LEVEL <unsigned_integer>
| TIMEOUT <unsigned_integer>
```

parameter_name [Page 43], password [Page 42], user_name [Page 40], unsigned_integer [Page 34]

Explanation

If the parameter name/user name and parameter name/password combination is valid, the user [Page 25] opens a database session and gains access to the database. As a result, he or she is the current user in this session.

A transaction is implicitly opened.

Each CONNECT option (`connect_option`) may only be specified once.

SQLMODE

The specification `SQLMODE <INTERNAL|ANSI|DB2|ORACLE>` can be used to select the SQL mode [Page 29]. The default SQL mode is INTERNAL.

The CONNECT option `SQLMODE <INTERNAL|ANSI|DB2|ORACLE>` is not allowed in programs. The appropriate precompiler option must be used to specify an SQLMODE other than INTERNAL.

ISOLATION LEVEL

The isolation level specified in the CONNECT statement is applied to each new transaction. This specification determines the lock operation type (Lock Behavior).

The isolation level is set using an integer without a plus/minus sign after the keywords ISOLATION LEVEL. The following values are permissible: 0, 1, 2, 3, 10, 15, 20, and 30. If no isolation level is specified, isolation level 1 is used.

- Isolation Level 0
- Isolation Level 1 or 10
- Isolation Level 15
- Isolation Level 2 or 20
- Isolation Level 3 or 30

TIMEOUT

The specified timeout value must be smaller than or the same as the current timeout value, otherwise an error is returned at logon.

If no timeout value is specified, the system uses the current timeout value.

If the TIMEOUT value is set to 0, the inactivity period is not monitored. This can result in a situation where database resources are not available again even though the associated application was concluded or aborted without a RELEASE statement [Page 217].

Users with the NOT EXCLUSIVE attribute

Users defined with the attribute NOT EXCLUSIVE can open several sessions at the same time. Whenever this is the case, or whenever two users of the same user group open a session at the same time, the sessions are considered to be distinct. This means that lock requests of the sessions concerned can collide.

SET Statement (set_statement)

The SET statement alters the properties of a database session [Page 27].

Syntax

```
<set_statement> ::= SET ROLE ALL [EXCEPT <role_name>] | SET ROLE NONE
| SET ROLE <role_name> [IDENTIFIED BY <password>]
| SET ISOLATION LEVEL <unsigned_integer>
```

role_name [Page 45], password [Page 42], unsigned_integer [Page 34]

Explanation

SET ROLE

DEFAULT ROLE in the ALTER USER statement [Page 165] or ALTER USERGROUP statement [Page 166] specifies which of the roles [Page 25] assigned to the current user or user group is active in the user session or group member session. If a role is active, the current user has all the privileges that are included in the role.

If a role that is activated automatically when a session is opened is assigned to the current user with the ALTER USER statement or ALTER USERGROUP statement, it is deactivated when the SET statement is executed if it is not identified by the SET ROLE specification in the SET statement.

- **ALL**: all roles assigned to the current user are active. EXCEPT can be used to exclude specified roles from activation.
- **NONE**: none of the roles is active.
- **Role name specified**: the roles specified here must exist and be assigned to the current user. If a password exists for the role, it must be defined in the SET statement

in addition to the owner of the role.
The role identified with role name is activated.

ISOLATION LEVEL

Specifying an isolation level changes the lock behavior for all subsequent SQL statements of the current database session.

The isolation level is set using an integer without a plus/minus sign after the keywords ISOLATION LEVEL. The following values are permissible: 0, 1, 2, 3, 10, 15, 20, and 30.

- Isolation Level 0
- Isolation Level 1 or 10
- Isolation Level 15
- Isolation Level 2 or 20
- Isolation Level 3 or 30

COMMIT Statement (commit_statement)

A COMMIT statement (`commit_statement`) terminates the current transaction and starts a new one (see transactions [Page 210]).

Syntax

```
<commit_statement> ::= COMMIT [WORK]
```

lock_statement [Page 215]

Explanation

The COMMIT statement terminates the current transaction. This means that the modifications executed within the transaction are recorded and are thus visible to concurrent users as well. The locks assigned to the transaction are released.

The COMMIT statement implicitly opens a new transaction. Any locks set within the new transaction are assigned to this transaction. The setting of locks in the new transaction is controlled by the isolation level declared in the CONNECT statement [Page 211] (see Isolation Level).

ROLLBACK Statement (rollback_statement)

The ROLLBACK statement cancels the current transaction [Page 210] and starts a new transaction.

Syntax

```
<rollback_statement> ::= ROLLBACK [WORK]
```

lock_statement [Page 215]

Explanation

The ROLLBACK statement cancels the current transaction. This means that any modifications made within the transaction are reversed. The locks assigned to the transaction are released.

The ROLLBACK statement implicitly opens a new transaction. Any locks set within the new transaction are assigned to this transaction. The setting of locks in the new transaction is

controlled by the isolation level declared in the CONNECT statement [Page 211] (see Isolation Level).

All result tables generated within the current transaction are implicitly deleted at the end of the transaction using the ROLLBACK statement.

SUBTRANS Statement (subtrans_statement)

The SUBTRANS statement divides a transaction [Page 210] into units known as subtransactions [Page 27].

Syntax

```
<subtrans_statement> ::= SUBTRANS BEGIN | SUBTRANS END | SUBTRANS
ROLLBACK
```

Explanation

SUBTRANS BEGIN

A subtransaction is opened, that is, the database records the current point in the transaction. This can be followed by any sequence of SQL statements. If this sequence does not contain an additional SUBTRANS BEGIN, all database modifications performed since the SUBTRANS BEGIN can be reversed using a SUBTRANS ROLLBACK.

The sequence, however, can also contain additional SUBTRANS BEGIN statements that open additional subtransactions. This means several nested subtransactions may be open at the same time.

SUBTRANS END

A subtransaction is closed, that is, the database system "forgets" the point in the transaction recorded with SUBTRANS BEGIN. An open subtransaction must exist for this purpose. If more than one open subtransaction exists, the last opened subtransaction is closed; that is, it is no longer considered to be an open subtransaction.

SUBTRANS ROLLBACK

SUBTRANS ROLLBACK reverses all database modifications performed within a subtransaction and then closes the subtransaction. Any database modifications performed by any subtransactions within the subtransaction are reversed, irrespective of whether they were ended with SUBTRANS END or SUBTRANS ROLLBACK. All result tables generated within the subtransaction are closed.

An open subtransaction must exist for this purpose. If more than one open subtransaction exists, the last opened subtransaction is rolled back. The subtransaction concerned is then no longer considered open.

Further information

The SUBTRANS statement does not affect locks assigned to the transaction. In particular, SUBTRANS END and SUBTRANS ROLLBACK do not release any locks.

The SUBTRANS statement is particularly useful in keeping the effects of subroutines or database procedures [Page 28] atomic; that is, it ensures that they either fulfil all their tasks or else have no effect. To this end, a SUBTRANS BEGIN is issued initially. If the subroutine succeeds in fulfilling its task, it is ended with a SUBTRANS END; in the event of an error, a SUBTRANS ROLLBACK is used to reverse all the modifications performed by the subroutine.

The COMMIT statement [Page 213] and the ROLLBACK statement [Page 213] close any open subtransactions implicitly.

LOCK Statement(lock_statement)

The LOCK statement assigns a lock to the current transaction (see lock).

Syntax

```
<lock_statement> ::=
LOCK [(WAIT)|(NOWAIT)] <lock_spec> IN SHARE MODE
LOCK [(WAIT)|(NOWAIT)] <lock_spec> IN EXCLUSIVE MODE
LOCK [(WAIT)|(NOWAIT)] <lock_spec> IN SHARE MODE <lock_spec> IN
EXCLUSIVE MODE
LOCK [(WAIT)|(NOWAIT)] <row_spec>... OPTIMISTIC

<lock_spec> ::= TABLE <table_name>,... | <row_spec>...
| TABLE <table_name>,... <row_spec>...
```

row_spec [Page 216], table_name [Page 47]

Explanation

The specified table cannot be a base table, a view table (see table [Page 23]), nor a synonym [Page 24]. If the table name identifies a view table, locks are set on the base tables on which the view table is based.

WAIT/NOWAIT

- If (NOWAIT) is specified, the database does not wait for a lock to be released by another transaction. Instead, it issues an error message if a lock collision occurs. If there is no collision, the requested lock is set.
- In the event of a lock collision, if either the WAIT option is omitted or (WAIT) is specified, the system waits for locks to be released, until the period specified by the database parameter REQUEST_TIMEOUT has elapsed.

<row_spec>...

A `<row_spec>...` creates a lock for the table row denoted by the key values or a position in a result table.

Specifying a `row_spec` requires that the specified table have a key column; that is, if the table name identifies a view table, this must be modifiable.

TABLE <table_name>,...

If `TABLE <table_name>,...` is specified, a lock is created for the table in question.

If the view table identified by the table name is not changeable, only a share lock can be set for this view table. As a result of this SQL statement, share locks are set for all base tables underlying the view table.

SHARE

SHARE defines a share lock for the listed objects. To set share locks, the current user must have the SELECT privilege.

EXCLUSIVE

EXCLUSIVE defines an exclusive lock for the listed objects. To set exclusive locks, the current user must have the UPDATE, DELETE, or INSERT privilege.

OPTIMISTIC

OPTIMISTIC defines an optimistic lock on rows. This is only meaningful in connection with isolation levels 0, 1, 10, and 15.

Deadlock

Whenever the database system recognizes a deadlock caused by locks, it ends the transaction with an implicit ROLLBACK WORK.

Reproducibility

If reproducible results are needed to read rows using a SELECT statement, the read objects must be locked and the locks must be kept until reproduction. Reproducibility usually requires that share locks are set for the tables concerned, either by explicitly locking them using one or more LOCK statements, or implicitly by using the isolation level 3. This ensures that other users cannot modify the table.

See also:

Lock Behavior

ROW specification (row spec)

The ROW specification (`row_spec`) is a syntax element in a LOCK statement [Page 215] or an UNLOCK statement [Page 216].

Syntax

```
<row_spec> ::= ROW <table_name> KEY <key_spec>,...
| ROW <table_name> CURRENT OF <result_table_name>
```

table_name [Page 47], result_table_name [Page 41]

Explanation

For tables defined without key columns, the implicit key column SYSKEY CHAR(8) BYTE can be used in a key specification.

If `CURRENT OF <result_table_name>` is specified, the result table must have been specified with FOR UPDATE.

UNLOCK Statement (unlock_statement)

The UNLOCK statement releases rows on rows.

Syntax

```
<unlock_statement> ::= UNLOCK <row_spec>... IN SHARE MODE
| UNLOCK <row_spec>... IN EXCLUSIVE MODE
| UNLOCK <row_spec>... IN SHARE MODE <row_spec>... IN EXCLUSIVE MODE
| UNLOCK <row_spec>... OPTIMISTIC
```

row_spec [Page 216]

Explanation

Using the UNLOCK statement, locks of the following types can be released within a transaction [Page 210] for single table rows that have not yet been changed: share locks, optimistic locks, and exclusive locks.

If a row has not been inserted, changed, or deleted, its exclusive lock cannot be released using the UNLOCK statement.

The UNLOCK statement does not fail if the specified lock does not exist or cannot be released.

See also:

Lock Behavior

RELEASE Statement (release_statement)

The RELEASE statement terminates a user's transaction [Page 210] and database session [Page 27].

Syntax

```
<release_statement> ::= COMMIT [WORK] RELEASE | ROLLBACK [WORK]
RELEASE
```

Explanation

Ending a session using a RELEASE statement implicitly deletes all result tables, the data stored in temporary base tables, and the metadata of these tables.

COMMIT WORK RELEASE

The current transaction is aborted without opening a new one. The user session is ended.

If the database system has to reverse the current transaction implicitly, COMMIT WORK RELEASE fails, and a new transaction is opened. The user session is not ended in this case.

ROLLBACK WORK RELEASE

The current transaction is aborted without opening a new one. Any database modifications performed during the current transaction are undone. The user session is ended. ROLLBACK WORK RELEASE has the same effect as a ROLLBACK statement [Page 213] followed by COMMIT WORK RELEASE.

If the accounting function in the database system is activated, information on the session is added to the table SYSACCOUNT.

Statistics

The database system addresses disks in units of 8KB. The term ‘page’ is used to refer to these units.

SQL statements for statistics management

UPDATE STATISTICS statement [Page 217]	MONITOR statement [Page 219]

UPDATE STATISTICS Statement (update_statistics_statement)

The UPDATE STATISTICS statement defines the storage requirements of tables and indexes as well as the value distribution of columns, and stores this information in the database catalog.

Syntax

```
<update statistics statement> ::=
  UPDATE STAT[ISTICS] COLUMN <table name>.<column name> [ESTIMATE
[<sample definition>]]
| UPDATE STAT[ISTICS] COLUMN (<column name>,...) FOR <table name>
[ESTIMATE [<sample definition>]]
| UPDATE STAT[ISTICS] COLUMN (*) FOR <table name> [ESTIMATE
[<sample definition>]]
| UPDATE STAT[ISTICS] <table name> [ESTIMATE [<sample definition>]]
| UPDATE STAT[ISTICS] [<owner>.][<identifier>]* [ESTIMATE
[<sample definition>]]
```

table_name [Page 47], column_name [Page 46], sample_definition [Page 117], owner [Page 41], identifier [Page 36]

Explanation

When the UPDATE STATISTICS statement is executed, information on the table, such as the number of rows, the number of pages used, the sizes of indexes, the value distribution within columns or indexes, and so on, is stored in the database catalog. These values are used by the Optimizer to determine the best strategy for executing SQL statements.

You will find information on the Optimizer functions in Optimizer: SAP DB 7.4.

The UPDATE STATISTICS statement implicitly performs a COMMIT statement [Page 213] for each base table; i.e. the transaction within which the UPDATE STATISTICS statement has been executed is closed.

When a CREATE INDEX statement [Page 148] is executed, the above-mentioned information is stored in the database catalog for the index as well as for the base table for which this index is being defined. No information is stored for other indexes defined on this base table.

The statistical values stored in the database catalog can be retrieved by selecting the system table OPTIMIZERSTATISTICS. Each row of the table describes statistical values of indexes, columns or the size of a table.

For a definition of the statistics system tables, see *System Tables: SAP DB 7.4*, Definition of the System Tables.

<table name>

If a table name is specified, the table must be a non-temporary base table and the user must have a privilege for it.

<column name>

If a column name is specified, this column must exist in specified table.

If * is specified, all columns in the table are assumed.

<identifier>*

Specifying `<identifier>*` has the same effect as issuing the UPDATE STATISTICS statement for all base tables for which the current user has a privilege, and whose table name begins with the `identifier`.

UPDATE STATISTICS *

The SYSDBA user can use UPDATE STATISTICS * to execute the UPDATE STATISTICS statement for all base tables, even if the SYSDBA has not been assigned a privilege for these tables.

ESTIMATE

- If ESTIMATE and a `sample definition` are specified, the database system estimates the statistical values by selecting data at random. The number of random selects can be given as number of rows or as percentage. For a specification of 50% or more, all rows are analyzed. The runtime of the UPDATE STATISTICS statement can be considerably reduced by specifying ESTIMATE. In most cases, the precision of the statistical values determined is sufficient.
- If ESTIMATE is specified without a `sample definition`, the database system estimates the statistical values by selecting data at random. The number of selects was defined with the CREATE TABLE statement or an ALTER TABLE statement by means of a sample_definition [Page 117] for the specified table. For a specification of 50% or more, all rows are analyzed. The runtime of the UPDATE STATISTICS statement can be considerably reduced by specifying ESTIMATE. In most cases, the precision of the statistical values determined is sufficient.
- If ESTIMATE is not specified, the database system determines exact statistical values by considering the complete data of the table. For large tables, the runtime can be considerably long.

MONITOR Statement (monitor_statement)

The MONITOR statement can be used to initialize counters for monitoring the database with 0.

Syntax

```
<monitor statement> ::= MONITOR INIT
```

Explanation

The database system always records result counters. These are initialized with 0 when the system is started. The MONITOR statement can be used to reset them to 0.

The event counters recorded by the database system can be retrieved by selecting monitor system tables.

For a definition of the monitor system tables, see *System Tables: SAP DB 7.4*, Definition of the System Tables.

Restrictions

Maximum values

Identifier length	32 characters
Precision of numeric values	38 digits
Number of tables	unlimited
Internal length of a table row	8088 bytes
Length of a LONG column	2147483647 bytes

Number of columns per table (with KEY)	1024
Number of columns per table (without KEY)	1023
Number of primary key columns per table	512
Sum of internal lengths of all key columns	1024 bytes
Number of indexes per table	255
Number of columns in an index	16
Sum of internal lengths of all columns belonging to an index	1024 bytes
Number of referential CONSTRAINT definitions (foreign key dependencies) per table	unlimited
Number of columns in a referential CONSTRAINT definition	16
Number of triggers per table	3
Number of rows per table	unlimited
Number of result columns in a SELECT instruction	1023
Number of join tables in a SELECT statement	64
Number of join conditions in a WHERE clause of a SELECT statement	128
Number of correlated columns in an SQL statement	64
Number of correlated columns in an SQL statement	16
Number of the columns in an ORDER or GROUP clause	128
Length of the columns in an ORDER or GROUP clause	1020 bytes
Number of parameters in an SQL statement	2000
Length of an SQL statement (Configuration parameter _PACKET_SIZE less administration overhead)	at least 16000 bytes

Syntax List

This documentation uses the BNF syntax notation with the following conventions:

	Explanation
KEYWORDS	Keywords are shown in uppercase letters for the sake of clarity. They can be entered in uppercase or lowercase letters.
<xyz>	Terms in angle brackets are placeholders for syntactical units explained in this document. Do not use angle brackets when entering an SQL statement.

clause ::= rule	Clauses are the building blocks of SQL statements. Rules describe how these building blocks are put together to form more complex clauses and also dictate the notation that is used.
$clause_1$ $clause_2$	The two clauses are written one after the other, separated by at least one blank.
[clause]	Optional clause. This clause can be ignored. Do not use square brackets when entering an SQL statement.
Clause1 | clause2 | ... | $clause_n$	Alternative clauses. You can use exactly one of these clauses.
Clause,...	The clause can be repeated as often as required. The individual repetitions must be written one after the other and separated by a comma and any number of blanks.
Clause...	The clause can be repeated as often as required. The individual repetitions must be written directly one after the other without a separating comma or blank.

The syntax rules are specified in the following form:

```
Clause ::=
  Rule
```

If you want an explanation of the syntax rules, you can use the clause link to go to the relevant part of the Reference Manual. As a result, you exit the syntax list itself.

A

```
<add_definition [Page 133]> ::=
  ADD <column_definition>,...
| ADD (<column_definition>,...)
| ADD <constraint_definition>
| ADD <referential_constraint_definition>
| ADD <key_definition>
```

```
<alias_name [Page 39]> ::=
  <identifier>
```

```
<all_function [Page 107]> ::=
  <set_function_name> ( [ALL] <expression> )
```

```
<alter_definition [Page 134]> ::=
  ALTER CONSTRAINT <constraint_name> CHECK <search_condition>
| ALTER <key_definition>
```

```
<alter_index_statement [Page 149]> ::=
  ALTER INDEX <index_name> [ON <table_name>] ENABLE
| ALTER INDEX <index_name> [ON <table_name>] DISABLE
| ALTER INDEX <index_name> [ON <table_name>] INIT USAGE
```

```
<alter_password_statement [Page 169]>::=
  ALTER PASSWORD <old_password> TO <new_password>
| ALTER PASSWORD <user_name> <new_password>

<alter_table_statement [Page 133]> ::=
  ALTER TABLE <table_name> <add_definition>
| ALTER TABLE <table_name> <drop_definition>
| ALTER TABLE <table_name> <alter_definition>
| ALTER TABLE <table_name> <column_change_definition>
| ALTER TABLE <table_name> <modify_definition>
| ALTER TABLE <table_name> <referential_constraint_definition>
| ALTER TABLE <table_name> DROP FOREIGN KEY
<referential_constraint_name>
| ALTER TABLE <table_name> <sample_definition>

<alter_user_statement [Page 165]> ::=
  ALTER USER <user_name> [<user_mode>]
    [TIMEOUT <unsigned_integer> | TIMEOUT NULL]
    [COSTWARNING <unsigned_integer> | COSTWARNING NULL]
    [COSTLIMIT <unsigned_integer> | COSTLIMIT NULL]
    [DEFAULT ROLE ALL [EXCEPT <role_name>] | DEFAULT ROLE NONE
    | DEFAULT ROLE <role_name> [IDENTIFIED BY <password>]]
    [[NOT] EXCLUSIVE]
    [DEFAULTCODE <ASCII | EBCDIC | UNICODE>]

<alter_usergroup_statement [Page 166]> ::=
  ALTER USERGROUP <usergroup_name> [<usergroup_mode>]
    [TIMEOUT <unsigned_integer> | TIMEOUT NULL]
    [COSTWARNING <unsigned_integer> | COSTWARNING NULL]
    [COSTLIMIT <unsigned_integer> | COSTLIMIT NULL]
    [DEFAULT ROLE ALL [EXCEPT <role_name>] | DEFAULT ROLE NONE
    | DEFAULT ROLE <role_name> [IDENTIFIED BY <password>]]
    [[NOT] EXCLUSIVE]
    [DEFAULTCODE <ASCII | EBCDIC | UNICODE>]

<argument [Page 152]> ::=
  <identifier>

<arithmetic_function [Page 73]> ::=
  TRUNC ( <expression>[, <expression>] )
| ROUND ( <expression>[, <expression>] )
| NOROUND ( <expression> )
| FIXED ( <expression>[, <unsigned_integer> [, <unsigned_integer] ] )
| FLOAT ( <expression>[, <unsigned_integer> ] )
| CEIL ( <expression> )
| FLOOR ( <expression> )
| SIGN ( <expression> )
| ABS ( <expression> )
| POWER ( <expression>, <expression> )
| EXP ( <expression> )
| SQRT ( <expression> )
| LN ( <expression> )
| LOG ( <expression>, <expression> )
| PI
| LENGTH ( <expression> )
| INDEX ( <string_spec>, <string_spec> [,<expression>[, <expression>]
] )

<assignment_statement [Page 154]> ::=
  SET <variable_name> = <expression>
```

B

<between_predicate [Page 56]> ::=
  <expression> [NOT] BETWEEN <expression> AND <expression>

<bool_predicate [Page 57]> ::=
  <column_spec> [IS [NOT] <TRUE | FALSE>]

<boolean_factor [Page 72]> ::=
  [NOT] <predicate>
| [NOT] (<search_condition>)

<boolean_term [Page 70]> ::=
  <boolean_factor>
| <boolean_term> AND <boolean_factor>

C

<call_statement [Page 183]> ::=
  CALL <dbproc_name> [(<expression>,...)] [WITH COMMIT]

<cascade_option [Page 132]> ::=
  CASCADE
| RESTRICT

<case_else_clause [Page 157]> ::=
  ELSE <statement>

<case_function [Page 99]> ::=
  <simple_case_function>
| <searched_case_function>

<case_statement [Page 154]> ::=
  <simple_case_statement>
| <searched_case_statement>

<character [Page 30]> ::=
  <digit>
| <letter>
| <extended_letter>
| <hex_digit>
| <language_specific_character>
| <special_character>

<close_statement [Page 208]> ::=
  CLOSE [<result_table_name>]

<column_attributes [Page 124]> ::=
  [<key_or_not_null_spec>] [<default_spec>] [UNIQUE]
[<constraint_definition>]
    [REFERENCES <referenced_table> [(<referenced_column>)]
[<delete_rule>]]

<column_change_definition [Page 133]> ::=
  COLUMN <column_name> NOT NULL
| COLUMN <column_name> DEFAULT NULL
| COLUMN <column_name> ADD <default_spec>
| COLUMN <column_name> ALTER <default_spec>
| COLUMN <column_name> DROP DEFAULT

```
<column_definition [Page 118]> ::=
  <column_name> <data_type> [<column_attributes>]
| <column_name> <domain_name> [<column_attributes>]

<column_list [Page 158]> ::=
  <column_name>
| <column_list>, <column_name>

<column_name [Page 46]> ::=
  <identifier>

<column_spec [Page 47]> ::=
  <column_name>
| <table_name>.<column_name>
| <reference_name>.<column_name>
| <result_table_name>.<column_name>

<comment [Page 150]> ::=
  <string_literal>
| <parameter_name>

<comment_on_statement [Page 150]> ::=
  COMMENT ON <object_spec> IS <comment>

<commit_statement [Page 213]> ::=
  COMMIT [WORK]

<comp_op [Page 59]> ::=
  <
| >
| <>
| !=
| =
| <=
| >=
| = | < | > (for machines with EBCDIC Code)
| ~= | ~< | ~> (for machines with ASCII Code)

<comparison_predicate [Page 58]> ::=
  <expression> <comp_op> <expression>
| <expression> <comp_op> <subquery>
| <expression_list> <equal_or_not> (<expression_list>)
| <expression_list> <equal_or_not> <subquery>

<connect_option [Page 211]> ::=
  SQLMODE <INTERNAL | ANSI | DB2 | ORACLE>
| ISOLATION LEVEL <unsigned_integer>
| TIMEOUT <unsigned_integer>

<connect_statement [Page 211]> ::=
  CONNECT <parameter_name> IDENTIFIED BY <parameter_name>
[<connect_option>...]
| CONNECT <parameter_name> IDENTIFIED BY <password>
[<connect_option>...]
| CONNECT <user_name> IDENTIFIED BY <parameter_name>
[<connect_option>...]
| CONNECT <user_name> IDENTIFIED BY <password> [<connect_option>...]

<constraint_definition [Page 127]> ::=
  CHECK <search_condition>
```

```
| CONSTRAINT <search_condition>
| CONSTRAINT <constraint_name> CHECK <search_condition>

<constraint_name [Page 40]> ::=
  <identifier>

<conversion_function [Page 103]> ::=
  NUM (<expression>)
| CHR (<expression>[,<unsigned_integer>])
| HEX (<expression>)
| HEXTORAW (<expression>)
| CHAR (<expression>[,<datetimeformat>])

<create_dbproc_statement [Page 152]> ::=
  CREATE DBPROC <procedure_name> [(<formal_parameter>,..)] [RETURNS
CURSOR] AS <routine>

<create_domain_statement [Page 140]> ::=
  CREATE DOMAIN <domain_name> <data_type> [<default_spec>]
[<constraint_definition>]

<create_index_statement [Page 148]> ::=
  CREATE [UNIQUE] INDEX <index_name> ON <table_name> (<column_name>
[ASC | DESC],...)

<create_role_statement [Page 169]> ::=
  CREATE ROLE <role_name> [IDENTIFIED BY <password>]

<create_sequence_statement [Page 140]> ::=
  CREATE SEQUENCE [<owner>.]<sequence_name>
    [INCREMENT BY <integer>]
    [START WITH <integer>]
    [MAXVALUE <integer> | NOMAXVALUE]
    [MINVALUE <integer> | NOMINVALUE]
    [CYCLE | NOCYCLE]
    [CACHE <unsigned_integer> | NOCACHE]
    [ORDER|NOORDER]

<create_synonym_statement [Page 142]> ::=
  CREATE [PUBLIC] SYNONYM [<owner>.]<synonym_name> FOR <table_name>

<create_table_statement [Page 115]> ::=
  CREATE TABLE <table_name> (<column_definition>
    [,<table_description_element>,...]) [IGNORE ROLLBACK]
[<sample_definition>]
| CREATE TABLE <table_name> [(<table_description_element>,...)]
    [IGNORE ROLLBACK] [<sample_definition>] AS <query_expression>
[<duplicates_clause>]
| CREATE TABLE <table_name> LIKE <table_name> [IGNORE ROLLBACK]

<create_table_temp> :: =
  <create_table_statement> for creating temporary tables,
    that is, the table name table_name in the CREATE TABLE statement
    must have the format TEMP.<identifier>.

<create_trigger_statement [Page 158]> ::=
  CREATE TRIGGER <trigger_name> FOR <table_name> AFTER
<trigger_event,..>
    EXECUTE (<routine>) [WHENEVER <search_condition>]

<create_user_statement [Page 160]> ::=
  CREATE USER <user_name> PASSWORD <password>
    [<user_mode>]
```

```
    [TIMEOUT <unsigned_integer>]
    [COSTWARNING <unsigned_integer>]
    [COSTLIMIT <unsigned_integer>]
    [[NOT] EXCLUSIVE]
    [DEFAULTCODE <ASCII | EBCDIC | UNICODE>]
| CREATE USER <user_name> PASSWORD <password> LIKE <source_user>
| CREATE USER <user_name> PASSWORD <password> USERGROUP
<usergroup_name>
```

<create_usergroup_statement [Page 162]> ::=

```
  CREATE USERGROUP <usergroup_name>
    [<usergroup_mode>]
    [TIMEOUT <unsigned_integer>]
    [COSTWARNING <unsigned_integer>]
    [COSTLIMIT <unsigned_integer>]
    [[NOT] EXCLUSIVE]
    [DEFAULTCODE <ASCII | EBCDIC | UNICODE>]
```

<create_view_statement [Page 143]> ::=

```
  CREATE [OR REPLACE] VIEW <table_name> [(<alias_name>,...)]
    AS <query_expression> [WITH CHECK OPTION]
```

D

<data_typ [Page 119]> ::=

```
  CHAR[ACTER] [(<unsigned_integer>)] [ASCII | BYTE | EBCDIC |
UNICODE]
| VARCHAR [(<unsigned_integer>)] [ASCII | BYTE | EBCDIC | UNICODE]|
LONG [VARCHAR] [ASCII | BYTE | EBCDIC | UNICODE]
| BOOLEAN
| FIXED (<unsigned_integer> [,<unsigned_integer>])
| FLOAT (<unsigned_integer>)
| INT[EGER]
| SMALLINT
| DATE
| TIME
| TIMESTAMP
```

<date_function [Page 92]> ::=

```
  ADDDATE (<date_or_timestamp_expression>, <expression>)
| SUBDATE ( <date_or_timestamp_expression>, <expression> )
| DATEDIFF ( <date_or_timestamp_expression>,
<date_or_timestamp_expression> )
| DAYOFWEEK ( <date_or_timestamp_expression> )
| WEEKOFYEAR ( <date_or_timestamp_expression> )
| DAYOFMONTH ( <date_or_timestamp_expression> )
| DAYOFYEAR ( <date_or_timestamp_expression> )
| MAKEDATE ( <expression>, <expression> )
| DAYNAME ( <date_or_timestamp_expression> )
| MONTHNAME ( <date_or_timestamp_expression> )
```

<date_or_timestamp_expression [Page 95]> ::=

```
  <expression>
```

<datetimeformat [Page 50]> ::=

```
  EUR
| INTERNAL
| ISO
| JIS
| USA
```

<dbproc_name [Page 40]> ::=

```
  [<owner>.]<procedure_name>
```

<declare_cursor_statement [Page 185]> ::=
  DECLARE <result_table_name> CURSOR FOR <select_statement>

<default_predicate [Page 60]> ::=
  <column_spec> <comp_op> DEFAULT

<default_spec [Page 125]> ::=
  DEFAULT <literal >
| DEFAULT NULL
| DEFAULT USER
| DEFAULT USERGROUP
| DEFAULT DATE
| DEFAULT TIME
| DEFAULT TIMESTAMP
| DEFAULT TRUE
| DEFAULT FALSE
| DEFAULT TRANSACTION
| DEFAULT STAMP
| DEFAULT SERIAL[(<unsigned_integer>)]

delete_rule [Page 130]> ::=
  ON DELETE CASCADE
| ON DELETE RESTRICT
| ON DELETE SET DEFAULT
| ON DELETE SET NULL

<delete_statement [Page 181]> ::=
  DELETE [FROM] <table_name> [<reference_name>]
    [KEY <key_spec>,...] [WHERE <search_condition>]
| DELETE [FROM] <table_name> [<reference_name>]
    WHERE CURRENT OF <result_table_name >

<delimiter_token [Page 38]> ::=
  (|) | , | . | + | - | * | / | < | > | <> | != | = | <= | >=
| ¬= | ¬< | ¬> (for machines with EBCDIC code)
| ~= | ~< | ~> (for machines with ASCII code)

<derived_column [Page 193]> ::=
  <expression> [[AS] <result_column_name >]
| <result_column_name> = <expression>

<digit [Page 30]> ::=
  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

digit_sequence [Page 33]> ::=
  <digit>...

<distinct_function [Page 193]>::=
  <set_function_name> (DISTINCT <expression >)

<distinct_spec [Page 193]> ::=
  DISTINCT
| ALL

<domain_name [Page 41]> ::=
  [<owner>.]<identifier>

<double_quotes [Page 37]> ::=
  "

<drop_dbproc_statement [Page 158]> ::=
  DROP DBPROC <dbproc_name>

<drop_definition [Page 136]> ::=
  DROP <column_name>,... [<cascade_option >] [RELEASE SPACE]
| DROP (<column_name>,...) [<cascade_option>] [RELEASE SPACE]

```
| DROP CONSTRAINT <constraint_name>
| DROP PRIMARY KEY
```

<drop_domain_statement [Page 140]> ::=
```
  DROP DOMAIN <domain_name>
```

<drop_index_statement [Page 149]> ::=
```
  DROP INDEX <index_name> [ON <table_name>]
```

<drop_role_statement [Page 170]> ::=
```
  DROP ROLE <role_name>
```

<drop_sequence_statement [Page 141]> ::=
```
  DROP SEQUENCE [<owner>.]<sequence_name>
```

<drop_synonym_statement [Page 142]> ::=
```
  DROP [PUBLIC] SYNONYM [<owner>.]<synonym_name>
```

<drop_table_statement [Page 132]> ::=
```
  DROP TABLE <table_name> [<cascade_option>]
```

<drop_trigger_statement [Page 159]> ::=
```
  DROP TRIGGER <trigger_name> OF <table_name>
```

<drop_user_statement [Page 164]> ::=
```
  DROP USER <user_name> [<cascade_option>]
```

<drop_usergroup_statement [Page 165]> ::=
```
  DROP USERGROUP <usergroup_name> [<cascade_option>]
```

<drop_view_statement [Page 147]> ::=
```
  DROP VIEW <table_name> [<cascade_option>]
```

<duplicates_clause [Page 176]> ::=
```
  REJECT DUPLICATES
| IGNORE DUPLICATES
| UPDATE DUPLICATES
```

E

<equal_or_not [Page 59]> ::=
```
  <>
| =
| ¬ =  (for machines with EBCDIC code)
| ~=  (for machines with ASCII code)
```

<exists_predicate [Page 60]> ::=
```
  EXISTS <subquery >
```

<exists_table_statement [Page 139]> ::=
```
  EXISTS TABLE <table_name >
```

<explain_statement [Page 209]> ::=
```
  EXPLAIN [(<result_table_name>)] <query_statement>
| EXPLAIN [(<result_table_name>)] <single_select_statement >
```

<exponent [Page 34]> ::=
```
  [<sign>][[<digit>]<digit>]<digit>
```

<expression [Page 52]> ::=
```
  <term>
| <expression> + <term>
| <expression> - <term>
```

<expression_list [Page 52]> ::=
```
  (<expression>,...)
```

<extended_expression [Page 177]> ::=
  <expression>
| DEFAULT
| STAMP

<extended_letter [Page 31]> ::=
  #
| @
| $

<extended_value_spec [Page 49]> ::=
  <value_spec>
| DEFAULT
| STAMP

<extraction_function [Page 97]> ::=
  YEAR (<date_or_timestamp_expression >)
| MONTH (<date_or_timestamp_expression>)
| DAY (<date_or_timestamp_expression>)
| HOUR (<time_or_timestamp_expression >)
| MINUTE (<time_or_timestamp_expression>)
| SECOND (<time_or_timestamp_expression>)
| MICROSECOND (<expression>)
| TIMESTAMP (<expression>[, <expression>])
| DATE (<expression>)
| TIME (<expression>)

F

<factor [Page 54]> ::=
  [<sign>] <value_spec>
| [<sign>] <column_spec>
| [<sign>] <function_spec>
| [<sign>] <set_function_spec>
| <scalar_subquery>
| <expression>

<fetch_statement [Page 205]> ::=
  FETCH [FIRST | LAST | NEXT | PREV | <position> | SAME]
[<result_table_name>]
    INTO <parameter_spec>,...

<final_select [Page 186]> ::=
  <select_statement>

<first_character [Page 37]> ::=
  <letter>
| <extended_letter>
| <language_specific_character>

<first_password_character [Page 42]> ::=
  <letter>
| <extended_letter>
| <language_specific_letter>
| <digit>

<fixed_point_literal [Page 33]> ::=
  [<sign>]<digit_sequence>[.<digit_sequence>]
| [sign]<digit_sequence>.
| [sign].<digit_sequence>

<floating_point_literal [Page 33]> ::=
  <mantissa>E<exponent>
| <mantissa>e<exponent>

<formal_parameter [Page 152]> ::=
  IN <argument> <data_type>
| OUT <argument> <data_type>
| INOUT <argument> <data_type>

<from_clause [Page 196]> ::=
  FROM <from_table_spec>,...

<from_table_spec [Page 196]> ::=
  <table_name> [<reference_name>]
| <result_table_name> [<reference_name>]
| (<query_expression>) [<reference_name>]
| <joined_table>

<function_spec [Page 73]> ::=
  <arithmetic_function>
| <trigonometric_function>
| <string_function>
| <date_function>
| <time_function>
| <extraction_function>
| <special_function>
| <conversion_function>

G

<grant_statement [Page 170]> ::=
  GRANT <priv_spec>,... TO <grantee>,... [WITH GRANT OPTION]
| GRANT EXECUTE ON <dbproc_name> TO <grantee>,...
| GRANT SELECT ON <sequence_name> TO <grantee>,... [WITH GRANT
OPTION]

<grant_user_statement [Page 168]> ::=
  GRANT USER <granted_users> [FROM <user_name>] TO <user_name>

<grant_usergroup_statement [Page 168]> ::=
  GRANT USERGROUP <granted_usergroups> [FROM <user_name>] TO
<user_name>

<granted_usergroups [Page 168]> ::=
  <usergroup_name>,...
| *

<granted_users [Page 168]> ::=
  <user_name>,...
| *

<grantee [Page 171]> ::=
  PUBLIC
| <user_name>
| <usergroup_name>
| <role_name>

<group_clause [Page 199]> ::=
  GROUP BY <expression>,...

H

<having_clause [Page 200]> ::=
  HAVING <search_condition>

<hex_digit [Page 31]> ::=
  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```
| A | B | C | D | E | F
| a | b | c | d | e | f
```

<hex_digit_seq [Page 32]> ::=
  <hex_digit><hex_digit>
| <hex_digit_seq><hex_digit><hex_digit>

<hex_literal [Page 32]> ::=
  x''
| X''
| x'<hex_digit_seq>'
| X'<hex_digit_seq>'

<hours [Page 96]> ::=
  <expression>

I

<identifier [Page 36]> ::=
  <simple_identifier>
| <double quotes><special_identifier><double_quotes>

<identifier_tail_character [Page 37]> ::=
  <letter>
| <extended_letter>
| <language_specific_character>
| <digit>
| <underscore>

<if_statement [Page 154]> ::=
  IF <search_condition> THEN <statement> [ELSE <statement>]

<in_predicate [Page 61]> ::=
  <expression> [NOT] IN <subquery>
| <expression> [NOT] IN <expression_list>
| <expression_list> [NOT] IN <subquery>
| <expression_list> [NOT] IN (<expression_list>,...)

<index_name [Page 42]> ::=
  <identifier>

<indicator_name [Page 42]> ::=
  <parameter_name>

<initial_select [Page 186]> ::=
  <query_spec>

<insert_expression [Page 173]> ::=
  <extended_expression>
| <subquery>

<insert_statement [Page 173]> ::=
  INSERT [INTO] <table_name> [(<column_name>,...)]
    VALUES (<insert_expression>,...) [<duplicates_clause>]
| INSERT [INTO] <table_name> [(<column_name>,...)]
    <query_expression> [<duplicates_clause>]
| INSERT [INTO] <table_name> SET <set_insert_clause>,...
[<duplicates_clause>]

<integer [Page 34]> ::=
  [sign]<unsigned_integer>

J

<join_predicate [Page 62]> ::=
  <expression> [<outer_join_indicator>] <comp_op> <expression>
[<outer_join_indicator>]

<join_spec> ::=
  ON <search_condition>

<joined_table [Page 197]> ::=
  <from_table_spec> CROSS JOIN <from_table_spec>
| <from_table_spec> [INNER] JOIN <from_table_spec> <join_spec>
| <from_table_spec> [<LEFT|RIGHT|FULL> [OUTER]] JOIN
    <from_table_spec> <join_spec>

K

<key_definition [Page 131]> ::
  PRIMARY KEY (<column_name>,...)

<key_or_not_null_spec [Page 124]> ::=
  [PRIMARY] KEY
| NOT NULL [WITH DEFAULT]

<key_spec [Page 52]> ::=
  <column_name> = <value_spec>

<key_word [Page 35]> ::=
  <not_reserved_key_word>
| <reserved_keyword>

L

<language_specific_character [Page 31]> ::=
  <every letter that occurs in a Northern, Central, or Southern
European language
  and is not included in the <letter> list>
| <for UNICODE-enabled databases: every character that is not
included in the ASCII code list from 0 to 127>

<letter [Page 30]> ::=
  A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z

<like_expression [Page 64]> ::=
  <expression>
| '<pattern_element>...'

<like_predicate [Page 64]> ::=
  <expression> [NOT] LIKE <like_expression> [ESCAPE <expression>]

<literal [Page 31]> ::=
  <string_literal>
| <numeric_literal>

<local_variable [Page 153]> ::=
  <variable_name> <data_type>

<local_variable_list [Page 153]> ::=
  <local_variable>
| <local_variable_list>;<local_variable>

<local_variables [Page 153]> ::=
  VAR <local_variable_list>;

<lock_option [Page 203]> ::=
  WITH LOCK [(IGNORE)|(NOWAIT)] [EXCLUSIVE | OPTIMISTIC] [ISOLATION LEVEL <unsigned_integer>]

<lock_spec [Page 215]> ::=
  TABLE <table_name>,...
| <row_spec>...
| TABLE <table_name>,... <row_spec>...

<lock_statement [Page 215]> ::=
  LOCK [(WAIT)|(NOWAIT)] <lock_spec> IN SHARE MODE
| LOCK [(WAIT)|(NOWAIT)] <lock_spec> IN EXCLUSIVE MODE
| LOCK [(WAIT)|(NOWAIT)] <row_spec>... OPTIMISTIC
| LOCK [(WAIT)|(NOWAIT)] <lock_spec> IN SHARE MODE <lock_spec> IN EXCLUSIVE MODE

M

<mantissa [Page 34]> ::=
  <fixed_point_literal>

<mapchar_set_name [Page 42]> ::=
  <identifier>

<match_char [Page 66]> ::=
  Every character except
    %
  | X'1F'
  | <underscore>
  | X'1E'

<match_set [Page 66]> ::=
  <underscore>
| X'1E'
| <match_char>

<match_string [Page 65]> ::=
  %
| X'1F'

<minutes [Page 96]> ::=
  <expression>

<modify_definition [Page 137]> ::=
  MODIFY (<column_name> [<data_type>] [<column_attributes>]...)

<monitor_statement [Page 219]> ::=
  MONITOR INIT

N

<named_query_expression [Page 191]> ::=
  <named_query_term>
| <named_query_expression> UNION [ALL] <query_term>
| <named_query_expression> EXCEPT [ALL] <query_term>

<named_query_primary [Page 192]> ::=
  <named_query_spec>
| (<named_query_expression>)

<named_query_spec [Page 195]> ::=
  SELECT [<distinct_spec>] <result_table_name> (<select_column>,...)
<table_expression>

<named_query_term [Page 192]> ::=
  <named_query_primary>
| <named_query_term> INTERSECT [ALL] <query_primary>

<named_select_statement [Page 186]> ::=
  <named_query_expression> [<order_clause>] [<update_clause>]
[<lock_option>] [FOR REUSE]

<new_index_name [Page 149]> ::=
  <index_name>

<new_table_name [Page 138]> ::=
  <table_name>

<next_stamp_statement [Page 183]> ::=
  NEXT STAMP [INTO] <parameter_name>

<not_reserved_key_word [Page 36]> ::=

ACTION	ACTIVATE	ADABAS	ADD	ADDRESS
ADD_MONTHS	AFTER	ANALYZE	AND	ANSI
APPEND	ARCHIVE	AS	ASC	AT
AUTO	AUTOSAVE			
BACKUP_PAGES	BAD	BEFORE	BEGIN	BEGINLOA
BEGINPROC	BETWEEN	BLOCK	BLOCKSIZE	BOTH
BREAK	BUFFER	BUFFERPOOL	BWHIERACHY	BY
CACHE	CACHELIMIT	CACHES	CALL	CANCEL
CASCADE	CAST	CATALOG	CATCH	CHECKPOI
CLEAR	CLOSE	CLUSTER	COMMENT	COMMIT
COMPLETE	COMPUTE	CONFIG	CONNECT	CONSTRAI
CONTAINER	CONTINUE	COSTLIMIT	COSTWARNING	CREATE
CURRENT_DATE	CURRENT_SCHEMA	CURRENT_TIME	CURRENT_TIMESTAMP	CURRVAL
CURSOR	CYCLE			
DATA	DAYS	DB2	DBA	DBFUNCTI
DBPROC	DBPROCEDURE	DB_ABOVE_LIMIT	DB_BELOW_LIMIT	DECLARE
DEFAULTCODE	DEGREE	DESC	DESCRIBE	DEVICE
DIAGNOSE	DIMENSION	DISABLE	DIV	DO
DOMAIN	DROP	DSETPASS	DUPLICATES	DYNAMIC
EDITPROC	ELSE	ENABLE	END	ENDLOAD
ENDPROC	ERROR	ESCAPE	ESTIMATE	EUR
EVENT	EXCLUSIVE	EXECUTE	EXPLAIN	EXPLICIT
EXTRACT				
FACT	FALSE	FETCH	FILE	FLUSH
FORCE	FOREIGN	FORMAT	FREEPAGE	FVERSION

GET	GRANT			
HIGH	HOLD	HOURS		
IDENTIFIED	IF	IMPLICIT	IN	INCREMEN
INDEXNAME	INDICATOR	INFO	INIT	INITRANS
INOUT	INPROC	INSTANCE	INSTR	IS
ISO	ISOLATION			
JAVA	JIS			
KEEP				
LABEL	LANGUAGE	LAST_DAY	LEADING	LEVEL
LIKE	LOAD	LOCAL	LOCK	LOGFULL
LOGWRITER	LOG_ABOVE_LIMIT	LOW		
MAXTRANS	MAXVALUE	MEDIANAME	MEDIUM	MICROSEC
MIGRATE	MINUS	MINUTES	MINVALUE	MODE
MODIFY	MONITOR	MONTHS	MONTHS_BETWEEN	
NAME	NEW	NEW_TIME	NEXTVAL	NEXT_DAY
NLSSORT	NLS_DATE_FORMAT	NLS_DATE_LANGUAGE	NLS_LANGUAGE	NLS_SORT
NOCACHE	NOCYCLE	NOLOG	NOMAXVALUE	NOMINVAL
NONE	NOORDER	NOREWIND	NORMAL	NOSORT
NOWAIT	NUMBER	NVL		
OBID	OFF	ONLY	OPEN	OPTIMIST
OPTIMIZE	OPTION	OR	ORACLE	OUT
OUTER	OVERWRITE			
PACKAGE	PAGE	PAGES	PARAM	PARSE
PARSEID	PASSWORD	PCTFREE	PCTUSED	PERCENT
PING	PIPE	POS	PRECISION	PREPARE
PRIV	PRIVILEGES	PROC	PROCEDURE	PSM
PUBLIC				
QUICK				
RANGE	RAW	RAWTOHEX	READ	RECURSIV
REFERENCES	REGISTER	RELEASE	REMOTE	REMOVE
RENAME	RESOURCE	RESTART	RESTORE	RESTRICT
RESUME	RETURN	RETURNS	REUSE	REVOKE
ROLE	ROLLBACK	ROW	ROWNUM	ROWS
SAME	SAMPLE	SAPR3	SAVE	SAVEPOIN
SCHEMA	SECONDS	SEGMENT	SELECTIVITY	SEQUENCE
SERVERDB	SESSION	SHARE	SHUTDOWN	SIMILAR
SOUNDS	SQLID	SQLMODE	STANDARD	STANDBY
START	STARTPOS	STAT	STATE	STOP

STORAGE	SUBPAGES	SUBTRANS	SUSPEND	SWITCH
SYNCHRONIZE	SYNONYM	SYSDATE		
TABID	TABLESPACE	TAKEOVER	TAPE	TEMP
THEN	TIMEOUT	TOPIC	TO_CHAR	TO_DATE
TO_NUMBER	TRACE	TRAILING	TRIGGER	TRUE
TRY	TYPE			
UNIQUE	UNKNOWN	UNLOAD	UNLOCK	UNTIL
UNUSED	USA	USAGE	USERID	
VALIDPROC	VARCHAR2	VARYING	VERIFY	VERSION
VIEW	VOLUME	VSIZE	VTRACE	
WAIT	WHENEVER	WHILE	WORK	WRITE
WRITER				
YEARS				

<null_predicate [Page 67]> ::=
  <expression> IS [NOT] NULL

<numeric_literal [Page 33]> ::=
  <fixed_point_literal >
| <floating_point_literal>

O

<object_spec [Page 150]> ::=
  COLUMN <table_name>.<column_name>
| DBPROC[EDURE] <dbproc_name>
| DOMAIN <domain_name>
| FOREIGN KEY <table_name>.<referential_constraint_name>
| INDEX <index_name> ON <table_name>
| SEQUENCE <sequence_name>
| [PUBLIC] SYNONYM <synonym_name>
| TABLE <table_name>
| TRIGGER <trigger_name>.<table_name>
| USER <user_name>
| USERGROUP <usergroup_name>
| <parameter_name>

<old_index_name [Page 149]> ::=
  <index_name>

<old_table_name [Page 138]> ::=
  <table_name>

<open_cursor_statement [Page 204]> ::=
  OPEN <result_table_name>

<order_clause [Page 202]> ::=
  ORDER BY <sort_spec>,...

<outer_join_inidicator [Page 62]> ::=
  (+)

<owner [Page 41]> ::=
  <user_name>

```
| <usergroup_name>
| TEMP
```

P

<parameter_name [Page 43]> ::=
```
  :<identifier>
| :<identfier>(<identifier>)
| :<identfier>(.<identifier>.)
```

<parameter_spec [Page 48]> ::=
```
  <parameter_name> [<indicator_name>]
```

<password [Page 42]> ::=
```
  <identifier>
| <first_password_character>[<identifier_tail_character>...]
```

<pattern_element [Page 65]> ::=
```
  <match_string>
| <match_set>
```

<position [Page 205]> ::=
```
  POS (<unsigned_integer>)
| POS (<parameter_spec>)
| ABSOLUTE <integer>
| ABSOLUTE <parameter_spec>
| RELATIVE <integer>
| RELATIVE <parameter_spec>
```

<predicate [Page 55]> ::=
```
  <between_predicate>
| <bool_predicate>
| <comparison_predicate>
| <default_predicate>
| <exists_predicate>
| <in_predicate>
| <join_predicate>
| <like_predicate>
| <null_predicate>
| <quantified_predicate>
| <rowno_predicate>
| <sound_predicate>
```

<priv_spec [Page 171]> ::=
```
  ALL [PRIV[ILEGES]] ON [TABLE] <table_name>,...
| <privilege>,...  ON [TABLE] <table_name>,...
| <role_name>
```

<privilege [Page 43]> ::=
```
  INSERT
| UPDATE [(<column_name>,...)]
| SELECT [(<column_name>,...)]
| SELUPD [(<column_name>,...)]
| DELETE
| INDEX
| ALTER
| REFERENCES [(<column_name>,...)]
```

<procedure_name [Page 40]> ::=
```
  <identifier>
```

Q

<quantified_predicate [Page 67]> ::=
  <expression> <comp_op> <quantifier> <expression_list>
| <expression> <comp_op> <quantifier> <subquery>
| <expression_list> <equal_or_not> <quantifier>
(<expression_list>,...)
| <expression_list> <equal_or_not> <quantifier> <subquery>

<quantifier [Page 69]> ::=
  ALL
| SOME
| ANY

<query_expression [Page 189]> ::=
  <query_term>
| <query_expression> UNION [ALL] <query_term>
| <query_expression> EXCEPT [ALL] <query_term>

<query_primary [Page 190]> ::=
  <query_spec>
| (<query_expression>)

<query_spec [Page 192]> ::=
  SELECT [<distinct_spec>] <select_column>,... <table_expression>

<query_statement [Page 184]> ::=
  <declare_cursor_statement>
| <recursive_declare_cursor_statement>
| <named_select_statement>
| <select_statement>

<query_term [Page 190]> ::=
  <query_primary>
| <query_term> INTERSECT [ALL] <query_primary>

R

<recursive_declare_cursor_statement [Page 186]> ::=
  DECLARE <result_table_name> CURSOR FOR WITH RECURSIVE
<reference_name>   (<alias_name>,...) AS (<initial_select> UNION
ALL <recursive_select>) <final_select>

<recursive_select [Page 186]> ::=
  <query_spec>

<reference_name [Page 45]> ::=
  <identifier>

<referenced_column [Page 128]> ::=
  <column_name>

<referenced_table [Page 128]> ::=
  <table_name>

<referencing_column [Page 128]> ::=
  <column_name>

<referential_constraint_definition [Page 128]> ::=
  FOREIGN KEY [<referential_constraint_name>]
(<referencing_column>,...)
    REFERENCES <referenced_table> [(<referenced_column>,...)]
[<delete_rule>]

<referential_constraint_name [Page 44]> ::=
  <identifier>

<regular_token [Page 35]> ::=
  <literal>
| <key_word>
| <identifier>
| <parameter_name>

<release_statement [Page 217]> ::=
  COMMIT [WORK] RELEASE
| ROLLBACK [WORK] RELEASE

<rename_column_statement [Page 139]> ::=
  RENAME COLUMN <table_name>.<column_name> TO <column_name>

<rename_index_statement [Page 149]> ::=
  RENAME INDEX <old_index_name> [ON <table_name>] TO <new_index_name>

<rename_synonym_statement [Page 143]> ::=
  RENAME [PUBLIC] SYNONYM <old_synonym_name> TO <new_synonym_name>

<rename_table_statement [Page 138]> ::=
  RENAME TABLE <old_table_name> TO <new_table_name>

<rename_user_statement [Page 167]> ::=
  RENAME USER <user_name> TO <new_user_name>

<rename_usergroup_statement [Page 168]> ::=
  RENAME USERGROUP <usergroup_name> TO <new_usergroup_name>

<rename_view_statement [Page 147]> ::=
  RENAME VIEW <old_table_name> TO <new_table_name>

<reserved_key_word [Page 36]> ::=

ABS	ABSOLUTE	ACOS	ADDDATE	ADDTIME
ALL	ALPHA	ALTER	ANY	ASCII
ASIN	ATAN	ATAN2	AVG	
BINARY	BIT	BOOLEAN	BYTE	
CASE	CEIL	CEILING	CHAR	CHARACTER
CHECK	CHR	COLUMN	CONCAT	CONSTRAINT
COS	COSH	COT	COUNT	CROSS
CURDATE	CURRENT	CURTIME		
DATABASE	DATE	DATEDIFF	DAY	DAYNAME
DAYOFMONTH	DAYOFWEEK	DAYOFYEAR	DBYTE	DEC
DECIMAL	DECODE	DEFAULT	DEGREES	DELETE
DIGITS	DISTINCT	DOUBLE		
EBCDIC	EXCEPT	EXISTS	EXP	EXPAND
FIRST	FIXED	FLOAT	FLOOR	FOR
FROM	FULL			
GET_OBJECTNAME	GET_OWNER	GRAPHIC	GREATEST	GROUP
HAVING	HEX	HEXTORAW	HOUR	
IFNULL	IGNORE	INDEX	INITCAP	INNER

INSERT	INT	INTEGER	INTERNAL	INTERSECT
INTO				
JOIN				
KEY				
LAST	LCASE	LEAST	LEFT	LENGTH
LFILL	LINK	LIST	LN	LOCATE
LOG	LOG10	LONG	LONGFILE	LOWER
LPAD	LTRIM			
MAKEDATE	MAKETIME	MAPCHAR	MAX	MBCS
MICROSECOND	MIN	MINUTE	MOD	MONTH
MONTHNAME				
NATURAL	NCHAR	NEXT	NO	NOROUND
NOT	NOW	NULL	NUM	NUMERIC
OBJECT	OF	ON	ORDER	
PACKED	PI	POWER	PREV	PRIMARY
RADIANS	REAL	REJECT	RELATIVE	REPLACE
RFILL	RIGHT	ROUND	ROWID	ROWNO
RPAD	RTRIM			
SECOND	SELECT	SELUPD	SERIAL	SET
SHOW	SIGN	SIN	SINH	SMALLINT
SOME	SOUNDEX	SPACE	SQRT	STAMP
STATISTICS	STDDEV	SUBDATE	SUBSTR	SUBSTRING
SUBTIME	SUM	SYSDBA		
TABLE	TAN	TANH	TIME	TIMEDIFF
TIMESTAMP	TIMEZONE	TO	TOIDENTIFIER	TRANSACTION
TRANSLATE	TRIM	TRUNC	TRUNCATE	
UCASE	UID	UNICODE	UNION	UPDATE
UPPER	USER	USERGROUP	USING	UTCDATE
UTCDIFF				
VALUE	VALUES	VARCHAR	VARGRAPHIC	VARIANCE
WEEK	WEEKOFYEAR	WHEN	WHERE	WITH
YEAR				
ZONED				

<result_column_name [Page 193]> ::=
  <identifier>

<result_table_name [Page 41]> ::=
  <identifier>

<revoke_statement [Page 172]> ::=
  REVOKE <priv_spec>,... FROM <grantee>,... [<cascade_option>]
| REVOKE EXECUTE ON <dbproc_name> FROM <grantee>,...

```
| REVOKE SELECT ON <sequence_name> FROM <grantee>,...
[<cascade_option>]
```

<role_name [Page 45]> ::=
  <identifier>

<rollback_statement [Page 213]> ::=
  ROLLBACK [WORK]

<routine [Page 153]> ::=
  [<local_variables>] <statement_list>;

<routine_sql_statement [Page 154]> ::=
  <call_statement>
| <close_statement>
| <create_table_temp>
| <declare_cursor_statement>
| <delete_statement>
| <fetch_statement>
| <insert_statement>
| <lock_statement>
| <select_statement>
| <named_select_statement>
| <single_select_statement>
| <subtrans_statement>
| <update_statement>

<row_spec [Page 216]> ::=
  ROW <table_name> KEY <key_spec>,...
| ROW <table_name> CURRENT OF <result_table_name>

<rowno_column [Page 193]> ::=
  ROWNO [<result_column_name>]
| <result_column_name> = ROWNO

<rowno_predicate [Page 69]> ::=
  ROWNO < <unsigned_integer>
| ROWNO < <parameter_spec>
| ROWNO <= <unsigned_integer>
| ROWNO <= <parameter_spec>

S

<sample_definition [Page 117]> ::=
  SAMPLE <unsigned_integer> ROWS
| SAMPLE <unsigned_integer> PERCENT

<scalar_subquery [Page 201]> ::=
  <subquery>

<search_and_result_spec [Page 99]> ::=
  <search_expression>, <result_expression>

<search_condition [Page 70]> ::=
  <boolean_term>
| <search_condition> OR <boolean_term>

<searched_case_function [Page 101]> ::=
  WHEN <search_condition> THEN <result_expression>
    [...]
  [ELSE <default_expression>]
END

<searched_case_statement [Page 156]> ::=
  CASE

```
    <searched_case_when_clause>…
    [<case_else_clause>]
  END [CASE]
```

<searched_case_when_clause [Page 156]> ::=

```
  WHEN <search_condition> THEN <statement>
```

<seconds [Page 96]> ::=

```
  <expression>
```

<select_column [Page 193]> ::=

```
  <table_columns>
| <derived_column>
| <rowno_column>
| <stamp_column>
```

<select_statement [Page 188]> ::=

```
  <query_expression> [<order_clause>] [<update_clause>]
[<lock_option>]
    [FOR REUSE]
```

<sequence_name [Page 45]> ::=

```
  <identifier>
```

<set_function_name [Page 108]> ::=

```
  COUNT
| MAX
| MIN
| SUM
| AVG
| STDDEV
| VARIANCE
```

<set_function_spec [Page 105]> ::=

```
  COUNT (*)
| <distinct_function>
| <all_function>
```

<set_insert_clause [Page 178]> ::=

```
  <column_name> = <extended_value_spec>
```

<set_statement [Page 212]> ::=

```
  SET ROLE ALL [EXCEPT <role_name>]
| SET ROLE NONE
| SET ROLE <role_name> [IDENTIFIED BY <password>]
| SET ISOLATION LEVEL <unsigned_integer>
```

<set_update_clause [Page 180]> ::=

```
  <column_name> = <extended_expression >
| <column_name>,... = (<extended_expression>,...)
| (<column_name>,...) = (<extended_expression>,...)
| <column_name> = <subquery>
| (<column_name>,...) = <subquery>
```

<sign [Page 33]> ::=

```
  +
| -
```

<simple_case_function [Page 102]> ::=

```
  CASE <check_expression>,
    WHEN <search_expression> THEN <result_expression>
      [...]
    [ELSE <default_expression>]
  END
```

<simple_case_statement [Page 157]> ::=
  CASE <expression>
    <simple_case_when_clause>...
    [<case_else_clause>]
  END [CASE]

<simple_case_when_clause [Page 157]> ::=
  WHEN <literal>[, ...] THEN <statement>

<simple_identifier [Page 36]> ::=
  <first_character>[<identifier_tail_character>...]

<single_select_statement [Page 208]> ::=
  SELECT [<distinct_spec>] <select_column>,... INTO
<parameter_spec>,...
    FROM <from_table_spec>,...[<where_clause>] [<group_clause>]
    [<having_clause>] [<lock_option>]

<sort_spec [Page 202]> ::=
  <unsigned_integer> [ASC | DESC]
| <expression> [ASC | DESC]

<sound_predicate [Page 70]> ::=
  <expression> [NOT] SOUNDS [LIKE] <expression>

<source_user [Page 160]> ::=
  <user_name>

<special_character [Page 31]> ::=
  every character except
    <digit>
  | <letter>
  | <extended_letter>
  | <hex_digit>
  | <language_specific_character >
  | <character for the end of a line in a file>

<special_function [Page 99]> ::=
  VALUE (<expression>,<expression>,...)
| GREATEST (<expression>,<expression>,...)
| LEAST (<expression>,<expression>,...)
| DECODE
(<check_expression>,<search_and_result_spec>,...[,<default_expression
>])
| case_function

<special_identifier [Page 37]> ::=
  <special_identifier_character>...

<special_identifier_character [Page 37]> ::=
 Any character

<sql_comment [Page 111]> ::=
  /*<comment text>*/
| --<comment text>

<stamp_column [Page 193]> ::=
  STAMP [<result_column_name>]
| <result_column_name> = STAMP

<statement [Page 154]> ::=
  BEGIN <statement_list> END
| BREAK
| CONTINUE
| CONTINUE EXECUTION
| <if_statement>

```
| <while_statement>
| <assignment_statement>
| <case_statement>
| RETURN
| STOP (<expression> [,<expression>] )
| TRY <statement_list>; CATCH <statement>
| <routine_sql_statement>
```

<statement_list [Page 154]> ::=

```
  <statement>
| <statement_list> ; <statement>
```

<string_function [Page 81]> ::=

```
  <string_spec> || <string_spec>
| <string_spec> & <string_spec>
| SUBSTR (<string_spec>,<expression>[,<expression>])
| LFILL (<string_spec>,<string_literal>[,<unsigned_integer>])
| RFILL (<string_spec>,<string_literal>[,<unsigned_integer>])
| LPAD
(<string_spec>,<expression>,<string_literal>[,<unsigned_integer>])
| RPAD
(<string_spec>,<expression>,<string_literal>[,<unsigned_integer>])
| TRIM (<string_spec>[,<string_spec>])
| LTRIM (<string_spec>[,<string_spec>])
| RTRIM (<string_spec>[,<string_spec>])
| EXPAND (<string_spec>,<unsigned_integer>)
| UPPER (<string_spec>)
| LOWER (<string_spec>)
| INITCAP (<string_spec>)
| REPLACE (<string_spec>,<string_spec>[,<string_spec>])
| TRANSLATE (<string_spec>,<string_spec>,<string_spec>)
| MAPCHAR (<string_spec>[,<unsigned_integer>][,<mapchar_set_name>])
| ALPHA (<string_spec>[,<unsigned_integer>])
| ASCII (<string_spec>)
| EBCDIC (<string_spec>)
| SOUNDEX (<string_spec>)
| GET_OBJECTNAME (<string_literal)
| GET_OWNER (<string_literal>)
```

<string_literal [Page 32]> ::=

```
  ''
| '<character>...'
| <hex_literal>
```

<string_spec [Page 52]> ::=

```
  <expression >
```

<subquery [Page 200]> ::=

```
  (<query_expression>)
```

<subtrans_statement [Page 214]> ::=

```
  SUBTRANS BEGIN
| SUBTRANS END
| SUBTRANS ROLLBACK
```

<synonym_name [Page 46]> ::=

```
  <identifier>
```

T

<table_columns [Page 193]> ::=

```
  *
```

```
| <table_name>.*
| <reference_name>.*
```

<table_description_element [Page 115]> ::=

```
  <column_definition>
| <constraint_definition>
| <referential_constraint_definition>
| <key_definition>
| <unique_definition>
```

<table_expression [Page 195]> ::=

```
  <from_clause> [<where_clause>] [<group_clause>] [<having_clause>]
```

<table_name [Page 47]> ::=

```
  [<owner>.]<identifier>
```

<term [Page 52]> ::=

```
  <factor>
| <term> * <factor>
| <term> / <factor>
| <term> DIV <factor>
| <term> MOD <factor>
```

time_expression [Page 96]> ::=

```
  <expression>
```

<time_or_timestamp_expresion [Page 97]> ::=

```
  <expression>
```

<time_function [Page 95]> ::=

```
  ADDTIME (<time_or_timestamp_expression>, <time_expression> )
| SUBTIME ( <time_or_timestamp_expression>, <time_expression> )
| TIMEDIFF ( <time_or_timestamp_expression>,
<time_or_timestamp_expression> )
| MAKETIME (<hours>, <minutes>, <seconds> )
```

<trigger_event [Page 158]> ::

```
  INSERT
| UPDATE [(<column_list>)]
| DELETE
```

<trigger_name [Page 47]> ::=

```
  <identifier>
```

<trigonometric_function [Page 80]> ::=

```
  COS ( <expression> )
| SIN ( <expression> )
| TAN ( <expression> )
| COT ( <expression> )
| COSH ( <expression> )
| SINH ( <expression> )
| TANH ( <expression> )
| ACOS ( <expression> )
| ASIN ( <expression> )
| ATAN ( <expression> )
| ATAN2 ( <expression>, <expression> )
| RADIANS ( <expression> )
| DEGREES ( <expression> )
```

U

<underscore [Page 37]> ::=

```
  _
```

<unique_definition [Page 132]> ::=
  [CONSTRAINT <index_name>] UNIQUE (<column_name>,...)

<unlock_statement [Page 216]> ::=
  UNLOCK <row_spec>... IN SHARE MODE
| UNLOCK <row_spec>... IN EXCLUSIVE MODE
| UNLOCK <row_spec>... IN SHARE MODE <row_spec>... IN EXCLUSIVE MODE
| UNLOCK <row_spec>... OPTIMISTIC

<unsigned_integer [Page 34]> ::=
  <numeric_literal>

<update_clause [Page 203] > ::=
  FOR UPDATE [OF <column_name>,...] [NOWAIT]

<update_statement [Page 178]> ::=
  UPDATE [OF] <table_name> [<reference_name>] SET
<set_update_clause>,...
    [KEY <key_spec>,...] [WHERE <search_condition>]
| UPDATE [OF] <table_name> [<reference_name>] (<column_name>,...)
    VALUES (<extended_value_spec>,...) [KEY <key_spec>,...]
    [WHERE <search_condition>]
| UPDATE [OF] <table_name> [<reference_name>] SET
<set_update_clause>,...
    WHERE CURRENT OF <result_table_name>
| UPDATE [OF] <table_name> [<reference_name>] (<column_name>,...)
    VALUES (<extended_value_spec>,...) WHERE CURRENT OF
<result_table_name>

<update_statistics_statement [Page 217]> ::=
  UPDATE STAT[ISTICS] COLUMN <table_name>.<column_name>
    [ESTIMATE [<sample_definition>]]
| UPDATE STAT[ISTICS] COLUMN (<column_name>,...) FOR <table_name>
    [ESTIMATE [<sample_definition>]]
| UPDATE STAT[ISTICS] COLUMN (*) FOR <table_name>
    [ESTIMATE [<sample_definition>]]
| UPDATE STAT[ISTICS] <table_name> [ESTIMATE [<sample_definition>]]
| UPDATE STAT[ISTICS] [<owner>.][<identifier>]* [ESTIMATE
[<sample_definition>]]

<user_mode [Page 162]> ::=
  DBA
| RESOURCE
| STANDARD

<user_name [Page 40]> ::=
  <identifier>

<usergroup_mode [Page 164]> ::=
  RESOURCE
| STANDARD

<usergroup_name [Page 39]> ::=
  <identifier>

V

<value_spec [Page 49] > ::=
  <literal>
| <parameter_spec>
| NULL
| USER
| USERGROUP
| SYSDBA

```
| UID
| [<owner >.]<sequence_name>.NEXTVAL
| [<owner>.]<sequence_name>.CURRVAL
| <table_name>.CURRVAL
| DATE
| TIME
| TIMESTAMP
| UTCDATE
| TIMEZONE
| UTCDIFF
| TRUE
| FALSE
| TRANSACTION
```

<variable_name [Page 154]> ::=
  <identifier>

<where_clause [Page 198]> ::=
  WHERE <search_condition>

<while_statement [Page 154]> ::=
  WHILE <search_condition> DO <statement>