

Database
Management
System

LINTER[®]

Version 5.9

Linter's SQL

Relational Expert Systems



Table of Contents

Introduction	9
General Elements	10
Characters	10
Unicode.....	10
Data Types	10
Character Data Type.....	11
CHAR, CHARACTER.....	11
CHAR VARYING, CHARACTER VARYING, VARCHAR.....	11
Byte.....	11
BYTE VARYING, VARBYTE.....	12
Unicode	12
NATIONAL CHAR, NARIONAL CHARACTER, NCHAR.....	12
NATIONAL CHAR VARYING, NATIONAL CHARACTER VARYING, NCHAR VARYING.....	12
Numeric Data Types	13
INTEGER, INT	13
BIGINT	13
SMALLINT	13
DECIMAL, DEC, NUMERIC.....	13
REAL.....	14
DOUBLE	14
Date	14
Boolean	14
BLOB	14
External File.....	15
Literals.....	15
Character Literal.....	15
Byte Literal.....	15
Unicode Literals.....	16
Numeric literals.....	16
Date literals.....	17
Boolean literal.....	17
Tokens	17
NULL.....	18
Value Specifcarion.....	18
USER.....	18
SYSDATE.....	19
LAST_ROWID.....	19
LAST_AUTOINC.....	19
ROWNUM	19
ROWID.....	19
ROWTIME.....	19
LINTER_NAME_LENGTH.....	19
PROC_PAR_NAME_LEN, PROC_INFO_SIZE, TRIGGER_INFO_SIZE, AUD_OBJ_NAME_LEN.....	20
DEFAULT	20

DATABASE.....	20
Database Objects	20
Tables, Columns, Rows	20
View	21
Indexes	21
Indexes and Keys.....	22
Users.....	22
Stored Procedure	22
Triggers.....	22
Sequences.....	22
Locks.....	22
Privileges	23
Roles.....	23
Transactions	23
Synonyms.....	23
Events.....	24
Integrity Constraint Conditions	24
Names.....	24
Linter Functions	26
Aggregate Functions.....	26
AVG	26
COUNT	27
DEFAULT	27
MAX	27
MIN	27
SUM.....	28
Linter SQL Scalar Functions	28
String Functions.....	28
String Concatenation	28
INITCAP	28
INSTR.....	29
LENGTH.....	29
OCTET_LENGTH	29
LOWER and UPPER	29
LPAD and RPAD.....	30
LTRIM and RTRIM.....	30
TRIM.....	30
SUBSTR.....	31
RIGHT_SUBSTR	31
REPEAT_STRING.....	31
INSERT	32
REPLACE	32
CHR.....	32
Numeric Functions	32
ABS	32
CEIL and FLOOR.....	33
MOD	33
RAND	33
Trigonometric Functions	33
Inverse Trigonometric Functions	34

Hyberbolic Functions	34
Exponent.....	34
Logarithmes (Ln, Log).....	34
Power	35
Rounding.....	35
Truncating	35
Sign	35
Square Root.....	36
List Processing Functions	36
Value Conversion Functions.....	36
Conversion to String	36
Conversion to Number.....	38
Conversion to Date	38
Conversion to Xml	39
Conversion to Hexadecimal	39
Conversion to Character String	39
RAW Conversion Functions.....	39
GETBYTE	39
GETWORD	40
GETLONG	41
GETBITS.....	41
GETSTR and GETRAW	41
BLOB Commands and Functions	42
COUNTBLOB.....	42
FINDBLOB	42
GETBLOB	43
LENBLOB.....	43
Date Functions	44
Date Arithmetic.....	44
DAYNAME.....	44
MONTHNAME	44
TO_GMTIME.....	44
TO_LOCALTIME.....	45
Miscellaneous Functions	45
DBNAME	45
NULLIFERROR.....	45
DECODE	46
NVL.....	46
SYS_GUID	46
SQL elements and conceptions	47
Column Specification.....	47
Expression	47
Cast.....	50
Case.....	51
Condition.....	52
Predicates	52
Comparison Predicates.....	53
BETWEEN Predicate	54

IN Predicate	55
LIKE Predicate.....	56
EXISTS Predicate	57
NULL Predicate	57
QUANTIFIED Predicate	58
Oracle Style Join Operator (+).....	58
Table Expressions	59
FROM	59
WHERE	60
GROUP BY.....	61
HAVING	62
ORDER BY	63
Joins	64
Query Specification.....	66
Query Expression	66
Fetch First	68
FOR UPDATE clause	68
FOR BROWSE	68
WAIT/NOWAIT	69
Subquery.....	69
Hierarchical Query	71
Data Definition Language (DDL).....	72
Create Table	72
Default Column Values	77
Drop Table	78
Create View	78
Drop View.....	79
Grant	80
Grant Execute	80
Revoke	81
Revoke Execute.....	82
Create Index	82
Drop Index.....	83
Alter Table.....	83
Rebuild	86
Press	86
Test Table	87
Truncate Table.....	87
Data Manipulation Language (DML)	89

Delete.....	89
Insert.....	89
Update.....	91
Massive Data Append.....	92
Start Append.....	92
End Append.....	93
Positioned Delete and Update.....	93
Positioned Deletion.....	93
Positioned Update.....	94
Lock and Unlock Table.....	95
Lock Table.....	95
Unlock Table.....	95
Transaction Management.....	96
Set Savepoint.....	96
Commit.....	96
Rollback.....	96
Read only.....	97
Triggers.....	97
Create Trigger.....	97
Drop Trigger.....	98
Alter Trigger.....	99
Stored Procedures.....	99
Create Procedure.....	99
Alter Procedure.....	99
Drop Procedure.....	100
Execute.....	100
Data Replication.....	101
Create Replication Server.....	101
Drop Replication Server.....	101
Create Replication Rule.....	101
Alter Replication Rule.....	102
Drop Replication Rule.....	103
Synchronization Replication Rule.....	104
Events.....	104
Create Event.....	104
Drop Event.....	105
Set Event.....	105
Wait Event.....	105
Get Event.....	106
Clear Event.....	106
Users And Roles.....	107
Create User.....	107
Drop User.....	107
Alter User.....	108
Create Role.....	108

Drop Role	108
Grant Role.....	108
Revoke Role	109
Synonyms	110
Create Synonym	110
Drop Synonym	110
User Queries Trace File Management.....	111
Transaction Control.....	112
Job Sharing Execution	113
Set User Priority.....	114
Sequences.....	115
Sequence Creation	115
Sequence Deletion	115
Sequence Use	116
Code Page Use.....	117
Code Page Creation	117
Set Code Page	118
Create Translation	119
Deleting Code Page	119
Deleting Code Page Translation	120
Linter's Extended Data Protection Features	121
Audit Statements	121
Start Audit	121
Stop Audit	121
Set Audit Parameters	121
Audit Control	121
Security Groups	122
Create Group.....	122
Rename Group.....	122
Group Access Grant.....	122
Revoke Group Access	122
Access Level.....	123
Create Level	123
Rename Level	123
Managing Workstation.....	123

Create Station	123
Drop Station.....	124
Grant Access on Unlisted Station	124
Revoke Access on Unlisted Station.....	124
Grant Access to Listed Station	124
Revoke Access to Listed Station	125
Devices	125
Create Device.....	125
Drop Device.....	125
Alter Device	125
Grant Access to Listed Device.....	126
Revoke Access to Listed Device	126
Grant Access on Unlisted Device	126
Revoke Access on Unlisted Device.....	126
Get Information about Security Tags	127

Introduction

SQL (Structured Query Language) is the most popular relational DBMS query language. It is used for writing queries in such DBMS as DB2, SYBASE ORACLE and Informix.

There are some important elements in SQL which are not implemented in SQL-92, such as triggers, stored procedures, UNICODE, set of scalar functions, and various extended security features and full-text search functions. Relx has implemented SQL-92 plus the key additional features found in SQL-99 standard. The scalar functions have been implemented in SQL Linter for compatibility with Oracle's SQL DB.

The syntax notation used in this document is BNF with some extensions:

◊	Angle brackets enclose the name of a syntactic element.
::=	Definition operator which divides the definite element from its definition.
[]	Square brackets indicate optional elements in a syntax clause.
{ }	Braces group elements into a syntax clause. An ellipsis indicates an element which may be repeated one or more times.
KEYWORDS	Are presented in sans serif capital letters. When used in commands, they are case insensitive.
	Vertical bar separates alternatives within brackets or braces.

Each component is described with some or all of the following and in the following order:

- 1) Description of the purpose of the element;
- 2) BNF description of the element's syntax;
- 3) Remarks and comments that describe syntactic rules not expressed in BNF and semantic rules and restrictions;
- 4) Cases that further illustrate the remarks.
- 5) Examples.

General Elements

Characters

Specify terminal characters of the language and string elements.

<Character>	::=	<Digit> <Letter> <Hexadecimal digit> <Special character>
<Digit>	::=	0 1 2 3 4 5 6 7 8 9
<Hexadecimal digit>	::=	<digit> A B C D E F a b c d e f
<Letter>	::=	<Uppercase> <Lowercase>
<Uppercase>	::=	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
<Lowercase>	::=	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
<Special>		A <Special character> is any character with a code from 0 to 255 other than <Digit> and <Letter> characters.

Unicode

The UNICODE is the universal character encoding standard used for representation of text for computer processing. Unicode uses 16-bit encoding that provides code for 65536 characters, including thousands of characters used in major languages of the world, all historic scripts and common national systems, punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, etc.

Data Types

A data type is a set of legal values. The logical representation of a value is a <Literal>.

A value is a primitive, basic, element that cannot be logically divided into smaller parts.

Values can be either null or non-null. A null value is a value that is distinct from all non-null values.

Specify data types.

<Data_Type>	::=	<Character Data Type> <UNICODE Data Type> < Numeric Data Type> <Date Data Type > <Boolean Data Type> <Blob Data Type > <External file Data Type>
-------------	-----	--

Character Data Type

Specify Character Type.

```
<Character Data Type> ::= {
    CHAR
  | CHARACTER
  | CHARVARYING
  | CHARACTER VARYING
  | VARCHAR
  | BYTE
  | BYTE VARYING
  | VARBYTE
  } (<length>)
```

A character or byte string is a sequence of one-byte characters within the code range of 0-255.

The syntax representation of a character string and a byte string is the same. There are two differences between character and byte strings in Linter's SQL:

When comparing: character strings are padded with spaces, byte strings are padded with zeroes.

When moving across software platforms with different character encoding: character strings are transformed, byte strings remain unchanged.

The result of the comparison of any character or byte string with undetermined value string (NULL) is False.

CHAR, CHARACTER

CHAR and CHARACTER are synonyms. CHAR is character string consisting of < length > characters. The value of < length > must be more than zero. By default, length is equal to 1. String length cannot exceed 4000. If the real length of a CHAR string is less than <length> then the CHAR string is padded with spaces to length except when the concatenation function is accomplished. In this case the CHAR string is not padded with spaces, moreover existing spaces are removed.

Character string comparison is determined by numeric comparison of the code for each byte in a pair having the same relative position in their respective strings. If the string lengths are different, the shorter string is padded on the right with spaces as necessary to reach the length of the longer string.

CHAR VARYING, CHARACTER VARYING, VARCHAR

CHAR VARYING, CHARACTER VARYING, and VARCHAR are synonyms. VARCHAR specifies a variable-length character string. <length> is maximum number of characters that can be hold in VARCHAR column and cannot exceed 4000. <length > must be specified. VARCHAR values are compared using a non-padded comparison semantic.

Byte

BYTE is a byte string consisting of <length> bytes. If the real length of BYTE string is less than <length > then BYTE string is padded with zeroes to length. Maximum length is 4000.

BYTE string comparison is similar to Char string comparison.

BYTE VARYING, VARBYTE

BYTE VARYING and VARBYTE are synonyms. VARBYTE specifies a variable-length byte string.

Unicode

Specify Unicode Type.

```
<UNICODE Data Type> ::= {
    NATIONAL CHAR
    | NATIONAL CHARACTER
    | NCHAR
    | NATIONAL CHAR VARYING
    | NATIONAL CHARACTER VARYING
    | NCHAR VARYING
    }(<length>)
```

The UNICODE is the universal character encoding standard used for representation of text for computer processing. Unicode uses 16-bit encoding that provides code for 65536 characters, including thousands of characters used in major languages of the world all historic scripts and common national systems, punctuation marks, diacritics, mathematical symbols, technical symbols, arrows, dingbats, etc.

NATIONAL CHAR, NATIONAL CHARACTER, NCHAR

NATIONAL CHAR, NATIONAL CHARACTER, NCHAR are synonyms. NCHAR is a UNICODE only data type. NCHAR specifies a character string of <length> characters of national character set. If <length> is omitted then the default length is 1 character (2 bytes). Every symbol in NCHAR string is represented by 2 bytes. The maximum length of NCHAR string cannot exceed 2000 characters.

Character string comparison for NCHAR data type is determined by comparison of the weight (an arbitrary number) of each character in a pair having the same relative position in their respective strings.

If the real length of NCHAR string is less than <length> then NCHAR string is padded with spaces to equal length.

The automatic conversion of upper to lower case is supported for Unicode strings.

A Unicode string may have undetermined value (NULL)

NATIONAL CHAR VARYING, NATIONAL CHARACTER VARYING, NCHAR VARYING

NATIONAL CHAR VARYING, NATIONAL CHARACTER VARYING, NCHAR VARYING are synonyms and UNICODE-only data types.

NCHAR VARYING specifies a variable-length character string. <length> is the maximum number of characters in an NCHAR VARYING string which cannot exceed 2000 characters.

Character string comparison for NCHAR VARYING data type is determined by comparison of the weight for each character in a pair having the same relative position in their respective strings.

Compare functions for NCHAR VARYING values use non-padded comparison semantic.

Numeric Data Types

The Numeric data type stores zero, positive and negative fixed and floating-point numbers.

Specify Numeric data type.

```
<Numeric data type> ::= { INTEGER | INT }
                        | BIGINT
                        | SMALLINT
                        | {DECIMAL | DEC | NUMERIC }
                        [( <precision> [, <scale> ] )]
                        | { REAL | DOUBLE } [ <precision> ]
```

INTEGER, INT

INTEGER and INT are synonyms. INT is a number with a precision of 10 and zero digits to the right of the decimal point (similar to long in the C programming language). INTEGER requires 4 bytes for storage.

The value range is -2147483648 to 2147483747.

Examples

```
64
-15
+100
32767
720176
12345678901
```

BIGINT

BIGINT is an exact number with a precision of 19 digits. A BIGINT value requires 8 bytes of storage.

The range for BIGINT value is -9223372036854775808 to +9223372036854775807.

SMALLINT

SMALLINT is a number with a precision of 5 and zero number of digits to the right of the decimal point (similar to short in C the programming language). SMALLINT requires 2 bytes for storage.

The value range is -32 768 to 32 767.

DECIMAL, DEC, NUMERIC

DECIMAL, DEC, NUMERIC are synonyms. DECIMAL is a number with a precision (maximum number of significant digits) equal to 30 and scale (maximum number of digits after the decimal point) equal to 10.

Examples

```

25.5
10000.
-15.
+37589.333333333

```

REAL

REAL is an approximate number with a precision of 6 (similar to float in the C programming language). Real requires 4 bytes for storage.

Specify REAL data type.

```

<REAL Number>      ::=  [+ |-]{<Mantissa>[E | e] <Exponent>}
<Mantissa>          ::=  [+ |-] {<Unsigned decimal numeral>
                           [.<Unsigned decimal numeral>]}
<Exponent>         ::=  [+ |-] {<Unsigned decimal numeral>}

```

The value range is $-1E+38$ to $1E+38$.

DOUBLE

Double is an approximate number with a precision of 15 (similar to double in the C programming language). Double requires 8 bytes for storage.

Specify DOUBLE data type.

```

<DOUBLE Number>    ::=  [+ |-]{<Mantissa>[E | e] <Exponent>}}
<Mantissa>          ::=  [+ |-] {<Unsigned decimal numeral>
                           [.<Unsigned decimal numeral>]}
<Exponent>         ::=  [+ |-] {<Unsigned decimal numeral>}

```

The value range is $-1 E+38$ to $1 E+38$.

Examples

```

15E1
2.E5
2.2e-1
+5.E+2

```

Date

Date values contain values for year, month, and day plus time values for hours, minutes, seconds, and ticks within the range 00.00.000-31.12.2099. The time value has a precision of one tick, 1/100 part of a second. Date values may be compared only with other date values.

Boolean

The Boolean data type is represented the logical values TRUE and FALSE.

BLOB

The BLOB (binary large object) data type is used to store unformatted binary information such as text and graphics. The size of the BLOB column may range from the 0 to 2GB.

External File

EXTFILE data type enables access to binary files stored in a file system outside the database.

```
<EXTFILE data> ::= EXTFILE (<file_name> [,<filter_name>])
```

Literals

Literal is a non-null data value.

```
<Literal> ::= <Character Literal>
           |<Byte Literal>
           |<UNICODE Literal >
           |<Numeric Literal>
           | <Date Literal>
           |<Boolean Literal>
```

Character Literal

Character Literal is a sequence of characters enclosed in apostrophes. The maximum length of character literal is 4000 characters.

```
<Character Literal> ::= '<Character>...'
<Character>        ::= <Nonquote character | <Quote symbol>
<Quote symbol>    ::= "
```


A <Nonquote character> is any character other than a quote character (').

To represent a quotation mark within a character literal enter two single quotation marks.

Data type of <Character Literal> is CHAR.

The length of <Character Literal> is number of characters in literal.

The length of <Character Literal> can be zero.

 For Linter version up to 5.9 Character literals is converted according Code Page CP 866 independently from current set Code Page (Variable LINTER_CP).

Byte Literal

Byte literal is a sequence of hexadecimal digits. The maximum length of byte literal is 4000 bytes.

```
<BYTE Literal> ::= HEX('<hexadecimal digit>...')
                | hex('<hexadecimal digit>...')
```

Linter allows using Byte literals to represent UNICODE literal, so you can transmit UNICODE literal as a two-byte value.

Data type of <Byte Literal> is BYTE.

Examples

```
Hex ( 'CFF0' )
```

```
hex('0A0D')

SELECT LENGTH(hex('00ffac0d'));
Result:
|      4 |
```

The length of <Byte Literal> can be zero.

```
SELECT length(hex(''));
Result:
|      0 |
```

Unicode Literals

UNICODE literal specifies representation of the character literal using the national character set.

```
<UNICODE literal> ::= N<Character Literal>
```

Numeric literals

```
<Numeric Literal> ::= <Exact number>
                    | <Approximate number>
<Exact number>    ::= [+ | -] {<Unsigned integer>
                    | <Unsigned integer>}
<Approximate number> ::= <Mantissa>[E | e <Exponent>]
<Mantissa>        ::= <Exact number>
<Exponent>        ::= [+ | -]<Unsigned integer>
<Unsigned integer> ::= {<Digit>...}
<Digit>           ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

If the decimal point of an <Exact number> is omitted, it is treated as following the last digit.

When the decimal point is omitted, the data type of an exact number is INTEGER (INT), SMALLINT or BIGINT.

The Data Type of exact number:

Range of values	Data Type
-32.768 to +32.767	SMALLINT
-2 147 483 647 to +2 147 483 648	INTEGER
-9 223 372 036 854 775 808 to 9 223 372 036 854 775 807	BIGINT

The data type of a fixed-point real number is DECIMAL (DEC, NUMERIC).

The data type of an approximate number is a floating point, single precision REAL, or double precision, DOUBLE, number.

The value of a numeric literal is calculated by the usual mathematical interpretation of numbers represented by decimals.

Date literals

Date literal specify.

<Date Literal>	::=	'<Date representation>'
<Date representation>	::=	<Format1> <Format2> <Format3> <Format4> <Format5>
<Format1>	::=	DD-MM-[CC]YY[:[h]h[:[m]i[:[s]s[.[t]t]]]]
<Format2>	::=	MM/DD/[CC]YY[:[h]h[: [m]i[:[s]s[.[t]t]]]]
<Format3>	::=	DD.MM.[CC]YY[:[h]h[:[m]i[:[s]s[.[t]t]]]]
<Format4>	::=	DD-MON.[CC]YY[:[h]h[:[m]i[:[s]s[.[t] t]]]]
<Format5>	::=	CCYY- MM -DD [: [h] h [: [m] i [: [s] s [.[t] t]]]

where:

- DD - day of month;
- MM - month of year;
- CC - century;
- YY - year of century;
- hh - hours of day;
- mi - minutes of hour;
- ss - second of minute;
- tt - ticks.

The data type of a DATE literal is DATE.

If the time part of a <DATE literal> is omitted the date has implied time of 0 hours 00 min. 00 sec. 00 ticks. If century digits are omitted and the number is MORE than 37 the 20th century (digits 19) is implicit. If century digits are omitted, and the number is LESS than 37 the 21st century (digits 20) is implicit.

Boolean literal

<Boolean literal> ::= TRUE | FALSE

Tokens

<Token>	::=	<Delimiting token> <Non-delimiting token>
<Delimiting token>	::=	<Character literal> <Date literal> , () < > . : = * + - / < > >= <= <comment>
<Non-delimiting token>	::=	<Identifier> <Keyword> <Numeric literal>
<Identifier >	::=	{ < Letter> _ \$ } [{<Digit> <Letter> _ \$ }...]
<comment>	::=	<format1> <format2>

```

<format1> ::= //[<letter>... ]<Line feed character
<format2> ::= /*[<letter>...]*/
    
```

<Token>, which is not a Character literal, must not contain a spaces.

The <Identifier> cannot contain more than 66 characters and cannot be identical to any keyword.

Examples

```

Name
$$$$S11
First_name
    
```

NULL

NULL values are undefined values. The value of NULL is interpreted as UNKNOWN, not FALSE, zero, or Anything Else.

Any arithmetic expression containing a NULL is always evaluated to NULL. All operators return NULL when given a NULL operand.

Nulls can appear in columns of any data type that are not restricted by NOT NULL or PRIMARY KEY integrity constraints.

Value Specificarion

Specifies a parameter or a variable value.

```

<Value specification> ::= <Literal>
| USER
| SYSDATE
| LAST_ROWID
| LAST_AUTOINC
| ROWNUM
| ROWID
| ROWTIME
| LINTER_NAME_LENGTH
| PROC_PAR_NAME_LEN |
| P ROC_I N FO_S IZ E
| TRIGGER_INFO_SIZE
| AU D_O B J_ N A M E_ L E N
| DEFAULT
    
```

<Value specification> specifies a value that is not actually stored in the table. You can select <Value specification>, but you cannot change their values.

USER

The value specified by the USER keyword is equal to the current Authorization identifier.

The data type of the USER value is CHAR(18).

SYSDATE

The value specified by the SYSDATE keyword is equal to the current date. The data type of the SYSDATE value is DATE.

LAST_ROWID

LAST_ROWID returns the value of ROWID returned when last INSERT, UPDATE, or DELETE statements were accomplished. Value of the LAST_ROWID has the data type INTEGER.

LAST_AUTOINC

LAST_AUTOINC returns the last value of AUTOINC column, added to any table during the current connection.

Value of the LAST_AUTOINC has the data type INTEGER.

ROWNUM

ROWNUM returns a number indicating the order in which row is selected from a table. For example the first row selected from a table has a ROWNUM of 1, second has 2 and so on. ROWNUM should be used only in SELECT statements.

Values of the ROWNUM have the data type INTEGER.

ROWID

ROWID returns row's number, uniquely identifies a row in the table. Values of the ROWID are unsigned, more than zero and have the data type INTEGER.

If you delete the row, its ROWID may be reassigned to a new row inserted later.

ROWID is the fastest way to access a single row.

ROWTIME

ROWTIME returns date and time of the last change of the row.

ROWTIME stores and represents date and time using Greenwich Mean Time.

The value of ROWTIME changes if UPDATE or UPDATE CURRENT statements have been accomplished.

LINTER_NAME_LENGTH

The value of LINTER_NAME_LENGTH is the maximum possible length of any identifier of a DB object.

PROC_PAR_NAME_LEN, PROC_INFO_SIZE, TRIGGER_INFO_SIZE, AUD_OBJ_NAME_LEN

PROC_PAR_NAME_LEN, PROC_INFO_SIZE, TRIGGER_INFO_SIZE, AUD_OBJ_NAME_LEN are constants which represent length of columns of system tables which have been changed from version to version. In general they are used to ensure each version is independent.

DEFAULT

If DEFAULT is supplied for column then the default value is supplied automatically when a new row is inserted and the value of column is omitted. If a default value is not explicitly defined for a column then the default for the column is implicitly set to NULL.

The data type of the default value must match the data type specified for the column. The column must be long enough to hold the default value.

DEFAULT is prohibited for columns BLOB and EXTFILE data types.

None of the columns in the primary key can have DEFAULT value.

DATABASE

A database is a set of data groped into tables.

A Linter database consists of system tables that contain the database dictionary and user tables that contain user data. System tables contain descriptions of database objects such as tables, table columns, users, and user privileges. The set of system tables is called the system database. No system table can be deleted.

After database generation, system tables belong to the user named SYSTEM. The SYSTEM user is the database manager, who can create new users and grant appropriate access levels.

Database Objects

Tables, Columns, Rows

A table is the basic unit of the Database. Table data is stored in rows and columns. A row is a sequence of values. Each row of the table has the same number of columns. A row is the smallest unit of data that can be inserted into the table or deleted from the table.

The table's description includes its name and its columns' descriptions. A description of the table also includes its file sizes, a cont of its occupied rows, and the maximum number of rows the table may hold.

The name of a table does not have to be unique in the entire database. However, each table belonging to an individual user must have a name that is unique to that user's list of tables.

Each column has a description and an ordered position within the table. Each column is given a column name, a data type, a data length (which could be predetermined by the data type), and the following characteristics:

default value	If any.
primary key status	Is the column the primary key?
unique status	Must a field value be unique within the column?
automatic increment status	Is the column value of each field to be automatically incremented?
index status	Is the column indexed?
null status	May the column values be NULL?

View

A VIEW appears to the end user just like a table. However, it is a virtual table and exists only during the request for it. A table, by contrast, is persistent, having a continuing existence in the DB with physically allocated file space.

A view is defined by a query that extracts data from the tables that the view references.

A view may be either updateable or read-only. You can update (insert or delete) data into updateable views but not into the read-only.

Views are very powerful because they let you tailor the presentation of data to different types of users.

Indexes

Indexes are structures actually stored in the database, which users create alter, and drop using SQL statements. You create an index to provide a fast access path to table data. A table column may be indexed or not. A column is indexed if an index structure was built for it. An index structure provides fast searching of rows via the indexed column.

Indexes can be unique or nonunique. Unique indexes guarantee that no two rows of a table have duplicate values in the key column (or columns). Nonunique indexes do not impose this restriction on the column values.

A composite index (also called a concatenated index) is an index that you create on multiple columns in a table. Columns in a composite index can appear in any order and need not be adjacent in the table. Composite indexes can speed retrieval of data for SELECT statements in which the WHERE clause references all or the leading portion of the columns in the composite index. Therefore the order of the columns used in the definition is important. Generally, the most commonly accessed or most selective columns go first.

An index is not a permanent characteristic of a column. Initially, no columns are indexed except for primary key columns. Indexes are logically and physically independent of the data in the associated table. You can create or drop an index at any time without affecting the base tables or other indexes.

Linter provides B-Tree indexes.

You can create 100 indexes for a table as long as the combination of columns differs for each index. You can create up to 10 indexes using the same column if you specify distinctly different combinations of the columns.

Indexes and Keys

Although the terms are often used interchangeably, there is a distinction between indexes and keys. Indexes are structures actually stored in the database, which users create, alter, and drop using SQL statements. You create an index to provide a fast access path to table data. Keys are strictly a logical concept. Keys correspond to another feature of Linter called integrity constraints, which enforce the business rules of a database.

Because Linter uses indexes to enforce some integrity constraints, the terms key and index are often used interchangeably. However, do not confuse them with each other.

Users

Each user with access to a database has a unique name (maximum 66 characters) and a 66-character password. The database user name need not be the same as the user's operating system account name.

Stored Procedure

A stored procedure is a group of Linter stored procedure language statements. A procedure is created and stored in compiled form in the database and can be run by a user or database application. See general information in document "Stored Procedure Functions".

Triggers

Triggers are specific stored procedures that run implicitly whenever a table or view is modified or when some user actions occur.

Sequences

A sequence is a database object that can generate unique sequential values.

Sequences are especially useful in multi-user environments for generating unique sequential numbers without the overhead of disk I/O or transaction locking.

Sequence numbers are used by SQL statements that reference the sequence. You can issue a statement to generate a new sequence number or use the current sequence number. After a statement in a user's session generates a sequence number, that particular sequence number is available only during that session. Each user that references a sequence has access to its own current sequence number.

Sequence numbers are generated independently of tables. Therefore the same sequence can be used for more than one table. Sequence number generation is useful to generate unique primary keys for your data automatically and to coordinate keys across multiple rows or tables.

Locks

Locks are mechanisms that prevent destructive interactions between transactions accessing the same resource - either user or system objects.

Privileges

A privilege is a right granted to a specified user to perform some action(s) on a specified table or view.

Each user has one of three access levels: CONNECT, RESOURCE, or DBA.

The user who created the table is the owner of that table. A DBA or RESOURCE access level is needed to create tables. A table's owner has all possible privileges on the table. The table's owner may grant some of these privileges to other users and revoke such grants.

The following privileges may be granted to a table:

SELECT	read data;
INSERT	insert rows;
UPDATE	update data;
DELETE	delete rows;
ALTER	change table properties;
INDEX	create and delete indices for table columns;
ALL	all the above privileges.

The table itself can only be deleted by the table's owner or by a user with DBA privileges.

Roles

Roles are used to control privilege management. Roles are named groups of related privileges that you grant to users or other roles.

Transactions

A transaction is a logical unit of work that contains one or more SQL statements. An error discovered in any operation will allow all operations in the transaction to be rolled back so that the state of the DB will be returned to what it was before the transaction was initiated.

A transaction is completed by use of Linter's Call commands COMT (from COMMIT) and RBAC (from ROLLBACK). If a transaction is finished by the COMMIT command, all updates made by this transaction are saved in the database and become accessible to all concurrent transactions. If a transaction is finished by the ROLLBACK command, all updates made by this transaction are cancelled (rolled back). Committed updates cannot be cancelled.

A transaction begins with the first executable SQL statement. A transaction ends when it is committed or rolled back, either explicitly with a COMMIT or ROLLBACK statement or implicitly when a DDL statement is issued.

Synonyms

Synonym is an alias for some DB objects. A synonym is not actually an object itself, but instead is a direct reference to an object.

Synonyms are used to mask the real name and owner of an object and simplify the SQL statement for DB users.

Events

Event-Linter mechanism provides the DB objects status monitoring in real time.

Integrity Constraint Conditions

Integrity constraints establish data integrity by restricting values in the base table. The constraints are checked after each transaction. A transaction that does not satisfy such constraints not performed and an exception code is returned to the application.

Linter provides the following integrity constraints:

NULL	Unless NULL is explicitly set, column fields may not contain null values.
AUTOINC	Ensures increments of each field in the column.
PRIMARY KEY	Prohibits insertion of a duplicate value or a null value in the specified column; also provides a method for distinguishing individual rows.
UNIQUE	Guarantees that only unique values will exist in the column.
DEFAULT	Supplies default values automatically when a new row is inserted and a value for the column is omitted.
CHECK	Is an integrity constraint on a column or a table that specifies a condition that every row of the table must satisfy.
FOREIGN KEY	Is usually the primary key of another table; also used to provide a method for creating implicit joins.
REFERENCES	Joins primary and foreign keys. References require that for each row of the table, the value in the foreign key matches a value in a primary key or is NULL.

Names

In Linter, the names of database objects, e.g., tables, users, cursors, and procedures, cannot exceed 66 characters. In the SQL Standard these are called identifiers and can be up to 128 characters in length.

Names can include digits, Latin letters, underscores (_) and dollar (\$) characters. A name must begin with a letter or '\$'.

The full name of a table may include the name of its owner. This is useful in allowing a non-owner access to the table owner's private synonym.

Specify names.

```
<Table name> ::= [<Authorization identifier>.]
                <Identifier>
```

<View name>	::=	[<Authorization identifier>.] <Identifier>
<Column name>	::=	[<Table name> <View name>.] <Identifier>
<Authorization identifier>	::=	<Identifier>
<User name>	::=	<Identifier>
<Trigger name>	::=	<Identifier>
<Code page name>	::=	<Identifier>
<Translation name>	::=	<Identifier>
<Role name>	::=	<Identifier>
<Index name>	::=	<Identifier>
<SQL-parameter name>	::=	<Identifier>
<Column specification>	::=	[<Qualifier>.]<Column name>
<Qualifier>	::=	<Table name> <Correlation name>
<Correlation name>	::=	<Identifier>
<Identifier>	::=	{<Letter> _ \$ } [{<Digit> <Letter> _ \$ }...]

<Table name> refers to a named table.

If the <Authorization Identifier> is omitted, the current session ser is assumed by default.

Two <Table name>s are equal only if their <Identifier's> are equal and their <Authorization Identifier's> are equal.

<Table name> is declared when the table is created.

<Column name> identifies the column name.

Identifiers, which are not keywords in SQL-89, SQL-92 standards may be represented as keywords only if they occur in specific commands.

Examples

```
SYSTEM.$$$USR,  
First_User.Auto,  
Second_User.Auto
```

Linter Functions

Aggregate Functions

Aggregate functions calculate a single value from a group of values. The group of values are acquired from a query using a SELECT statement, a HAVING and a GROUP BY clauses.

<Aggregate function>	::=	<Distinct function> <All function>
<Distinct function>	::=	{AVG MAX MIN SUM COUNT } (DISTINCT<Value expression>) DEFAULT(Column Name)
<All function>	::=	COUNT(*) {AVG MAX MIN SUM COUNT} ([ALL]<Value expression>)

Aggregate functions that include a DISTINCT - function modifier consider only distinct values of the argument expression. In contrast, the modifier ALL - function considers all values, including all duplicates.

The <Value expression> contained in a <Aggregate function> cannot contain an <Aggregate function>.

If a <Value expression> contains a <Column Specification> that is an outer reference <Value expression> cannot contain any other elements. See section "Column Specification".

If <Value expression> contains an outer reference <Column specification> the <Aggregate function> must be contained in the subquery of a HAVING clause.

AVG

Syntax

```
AVG ( [DISTINCT | ALL ] <Value expression> )
```

Usage

AVG returns the average value of <Value expression>. You can use AVG with Numeric data type columns only. AVG doesn't include rows where the expression is the NULL value. If you want to count rows with NULL value you have to substitute zero for the NULL value.

The Data type <Value expression > cannot be a Character type or Date type.

If data type of <Value expression> is an exact Numeric data type, the returned data type is DECIMAL.

If data type of <Value expression > is an approximate numeric type, the returned data type is DOUBLE.

Example

```
SELECT AVG(CASE I WHEN NULL THEN 0 ELSE I END) FROMTST;
```

COUNT

Syntax

COUNT (* | DISTINCT | ALL | <Value expression>)

Usage

The COUNT(*) argument with either DISTINCT or ALL is a table or a group of grouped tables. See the GROUP BY clause.

Counts the number of rows in a group depending on the specified parameters. COUNT never returns NULL. If you specify expression, COUNT returns the number of rows where expression is not NULL. If you specify the asterisk (*), COUNT returns all rows, including duplicates and NULLs.

DEFAULT

DEFAULT returns a column's default value.

Syntax

DEFAULT(<Column_Name>)

Usage

DEFAULT returns a column's default value.

If the column doesn't have a default value, a NULL is returned.

Example

```
SELECT id, default(color) FROM person GROUP BY id;
```

MAX

Syntax

MAX([DISTINCT | ALL]| <Value expression>)

Usage

Returns the maximum <Value expression> value found in each group of rows. Rows where <Value expression> is NULL are ignored. Returns NULL for a group containing no rows.

DISTINCT is included for completeness and doesn't influence the result of MAX function.

MIN

Syntax

MIN([DISTINCT | ALL]| <Value expression>)

Usage

Rows where <Value expression> is NULL are ignored. Returns NULL for a group containing no rows.

DISTINCT is included for completeness and doesn't influence the result of MIN function.

SUM

Syntax

```
SUM( DISTINCT | ALL <Value expression>)
```

Usage

Returns the total of the specified expression for each group of rows.

Rows where the specified expression is NULL are not included.

Returns NULL for a group containing no rows.

Data type of <Value expression > cannot be a Character type or Date type.

If data type of <Value expression> is an exact Numeric data type the returned data type is DECIMAL.

If data type of <Value expression> is approximate numeric type, the returned data type is DOUBLE.

Linter SQL Scalar Functions

The scalar functions implemented in Linter operate on column values and/or constants in expressions. They may be used anywhere in queries where it is possible to use expressions.

String Functions

String Concatenation

Combine two strings.

Syntax

```
<String1>||<String2> | <String1 >+<String2>
<String>          ::= <Character value expression>
```

Description

The result of this function is the string obtained by concatenation of two given strings. The data type of the result is CHAR with length equal to the sum of the lengths of the arguments.

INITCAP

Capitalize the first letter of each word in a string.

Syntax

```
INITCAP (<String>)
<String>  ::= <Character value expression>
```

Description

The result of this function is capitalization of the first letter of each word in the <String> argument. The data type of the result is the data type of the argument.

Remarks

Word separators within the <String> are any character which code less or equal than the cod of the Space character.

INSTR

Finds the start of a substring within a string.

Syntax

```
INSTR (<String>,<Substring>[,<Start>[,<Occurrence>] ] )  
<String>           ::= <Character value expression>  
<Substring>       ::= <Character value expression>  
<Start>           ::= <Numeric value expression>  
<Occurrence>     ::= <Numeric value expression>
```

Description

This function returns the position of the first character of <Substring> in the <String>, or zero if the substring is not found. The initial position for starting the search, <Start> and the number of occurrences, <Occurrence> may be specified. The data type of the result is INT.

LENGTH

Get the length of a string.

Syntax

```
LENGTH ( <String> )  
<String>           ::= <Character value expression>
```

Description

This function returns the length of the <String> argument without trailing blanks. The data type of the result is INT.

OCTET_LENGTH

Get the length of a string in bytes.

Syntax

```
OCTET_LENGTH (<String>)  
<String>           ::= <Character value expression>
```

Description

If data type of the <String> is CHAR or VARCHAR then the function returns the length of the <String> without trailing blanks.

If data type of the <String> is NCHAR or NCHAR VARYING then the function returns the L/2, where L is the length of the <String> in bytes.

The data type of the result is INT.

LOWER and UPPER

Convert string case.

Syntax

```
LOWER ( <String> )  
UPPER ( <String> )  
<String>           ::= <Character value expression>
```

Description

This function returns the string argument with all alphabetic characters converted into lowercase/uppercase. The data type of the result is the same as the data type of the argument.

LPAD and RPAD

Pad strings.

Syntax

```
LPAD ( <String>, <Length> [, <Pad string> ] )
RPAD ( <String>, <Length> [, <Pad string> ] )
<String>          ::= <Character value expression>
<Length>          ::= <Numeric value expression>
<Pad string>      ::= <Character value expression>
```

Description

This function returns the value of the string argument padded at the left (LPAD) or right (RPAD) with the character sequence specified by the <Pad string> argument, up to a length specified by <Length>. <Pad string> will be repeated as often as necessary to reach the necessary length. If <Pad string> is not included, the string is padded with blanks.

The data type of the result is CHAR with a length equal to the greater of <Length> or the length of the original string.

LTRIM and RTRIM

Remove string leading or trailing blanks or other padding characters.

Syntax

```
LTRIM ( <String> [ , <Substring> ] )
RTRIM ( <String> [ , <Substring> ] )
<String>          ::= <Character value expression>
<Substring>       ::= <Character value expression>
```

Description

This function returns the original string from which the character sequences given by the <Substring> argument at the left (LTRIM) or right (RTRIM) are removed.

If <Substring> is not included, spaces are removed by default.

The data type of the result is the same as the data type of the <String> argument.

TRIM

Remove character sequences from the beginning and end of the string.

Syntax

```
TRIM(<String>,<Substring>)
<String>          ::= <Character value expression>
<Substring>       ::= <Character value expression>
```

Description

This function returns the original strings from which the character sequences given by `<Substring>` argument at the left and right are removed.

If `<Substring>` is not included, spaces are removed by default.

SUBSTR

Syntax

```
SUBSTR ( <String>,<Start> [,<Length> ] )  
<String>      ::= <Character value expression>  
<Start>       ::= <Numeric value expression>  
<Length>      ::= <Numeric value expression>
```

Description

This function returns the substring extracted from `<String>`. The position within the string where the search is to start is specified by `<Start>`. The length of the substring to be extracted may specified by `<Length>`. If `<Length>` is omitted, the returned substring extends to the end of the string.

The data type of the result is the same as the data type of the `<String>` argument.

RIGHT_SUBSTR

Get specified number of characters from the end of the string.

Syntax

```
RIGHT_SUBSTR(<String>,<Length>)  
<String>      ::= <Character value expression>  
<Length>      ::= <Numeric value expression>
```

Description

This function returns the substring extracted from `<String>`.

The position within the string where the search is to start is the end of the string. The length of the substring to be extracted is specified by `<Length>`. If the length of the `<String>` is less than `<Length>` then the function returns original `<String>`.

REPEAT_STRING

Combine a specified number of the string.

Syntax

```
REPEAT_STRING (<String>,<Number>)  
<String>      ::= <Character value expression>  
<Number>      ::= <Numeric value expression>
```

Description

The result of this function is the string obtained by concatenation of `<Number>` of the original string. The data type of the result is CHAR with length equal to the sum of the lengths of the arguments.

INSERT

Syntax

```

INSERT(<String>,<Start>,<Length>,<Substring>)
<String>          ::= <Character value expression>
<Start>           ::= <Numeric value expression>
<Length>          ::= <Numeric value expression>
<Substring>       ::= <Character value expression>

```

Description

This function returns the string from which the character sequences specified by the <Length> argument are removed and the substring given by <Substring> is inserted into. The position within the string where the search is to start is specified by <Start>.

REPLACE

Syntax

```

REPLACE(<String>,<Substring1>,<Substring2>)
<String>          ::= <Character value expression>
<Substring1>      ::= <Character value expression>
<Substring2>      ::= <Character value expression>

```

Description

This function returns the original string in which character sequences given by <Substring1> is replaced by the character sequences given by <Substring2>.

CHR

Syntax

```

CHR(<Value>)
<Value>          ::= <Unsigned integer (0-127)>

```

Description

This function returns the character the CHAR representation of a Byte or <Space> in case of there isn't corresponding character in current Code page.

Example

```

Select chr(123) || to_char(sysdate) || chr(125) ;
Result :
|27-OCT-02|

```

Numeric Functions

ABS

Syntax

```

ABS ( <Numeric value expression > )

```

Description

The result of this function is the absolute value of the argument. The data type of the result is the same as the data type of the argument.

Example

```
SELECT ABS(5.7) ,  
ABS(4-65.2) ,  
ABS(AVG(salary)-MAX(salary)) From person;  
Result:  
| 5.7| 61.2| 40448.1643002029|
```

CEIL and FLOOR**Syntax**

```
CEIL(<Numeric value expression >)  
FLOOR(< Numeric value expression >)
```

Description

The result of this function is the value of the argument rounded up, CEIL, or down, FLOOR, to the nearest integer.

The data type of the result is DOUBLE.

MOD**Syntax**

```
MOD(<Numeric value expression1 >,<Numeric value expression2>)
```

Description

The data type of the <Numeric value expression's> should be REAL data type.

The result of this function is the remainder of division one Numeric value expression by another.

RAND**Syntax**

```
RAND([<Seed>])
```

Description

The result of this function is a random number in the interval 0 to 1 with an optional Seed. Seed is initialized value.

Trigonometric Functions**Syntax**

```
COS ( <Numeric value expression > )  
SIN ( <Numeric value expression > )  
TAN ( <Numeric value expression > )
```

Description

The result of this function is the cosine, sine, or tangent of the argument given in radians. The data type of the result is DOUBLE.

Inverse Trigonometric Functions**Syntax**

ACOS (<Numeric value expression>)
ASIN (<Numeric value expression>)
ATAN (<Numeric value expression>)
ATAN2 (<Numeric value expression1>, <Numeric value expression2>)

Description

The result of this function is the arc cosine, arc sine, or arc tangent of the argument given in radians. The data type of the result is DOUBLE. ATAN2 returns the arc tangent of <Numeric value expression1> and <Numeric value expression2>. Inputs are in an unbounded range, and outputs are in the range of $-\pi$ to π , depending on the signs of <Numeric value expression1> and <Numeric value expression2>, and are expressed in radians.

Hyberbolic Functions**Syntax**

COSH (<Numeric value expression>)
SINH (<Numeric value expression>)
TANH (<Numeric value expression>)

Description

The result of this function is the hyperbolic cosine, sine, or tangent of the argument. The data type of the result is DOUBLE.

Exponent**Syntax**

EXP (<Numeric value expression>)

Description

The result of this function is the exponent of the argument. The data type of the result is DOUBLE.

Logarithmes (Ln, Log)**Syntax**

EXP (<Numeric value expression>)
LOG (<Numeric value expression>)

Description

The result of this function is the natural or decimal logarithm of the argument. The data type of the result is DOUBLE.

Power

Syntax

```
POWER ( <Numeric value expression> ,<Exponent> )  
<Exponent>      ::=    <Numeric value expression>
```

Description

The result of this function is the <Numeric value expression> raised to the power specified by the <Exponent>. The data type of the result is DOUBLE.

Rounding

Syntax

```
ROUND ( <Numeric value expression> , <Precision> )  
<Precision>      ::=    <Numeric value expression>
```

Description

This function returns value rounded to <Precision> places right of the decimal point. <Precision> can be negative to round off digits left of the decimal point.

The data type of the result is DOUBLE.

Truncating

Syntax

```
ROUND ( <Numeric value expression>,<Precision> )  
<Precision>      ::=    <Numeric value expression>
```

Description

This function returns value truncated to <Precision> places. <Precision> can be negative to truncate <Precision> digits left of the decimal point.

The data type of the result is DOUBLE.

Sign

Syntax

```
SIGN ( <Numeric value expression> )
```

Description

The result of this function is the sign of the argument.

The result can be:

- -1, < Numeric value expression> – negative.
- 0, < Numeric value expression > – zero.
- 1, < Numeric value expression> – positive.

The data type of the result is INT.

Square Root

Syntax

```
SQRT ( <Numeric value expression> )
```

Description

The result of this function is a square root of the argument.

The data type of the result is DOUBLE.

List Processing Functions

A list is a sequence of values. There are two functions for working with lists: GREATEST returns the maximum value in a list; LEAST returns the minimum value in the list.

The data types of all values in the list must be compatible.

Syntax

```
GREATEST( <Value expression>[ ,... ] )
```

```
LEAST( <Value expression>[ ,... ] )
```

Description

The data type of the result is DOUBLE if the elements of the list have numeric types; otherwise the data type matches the data type of values in the list. If the list consists of character and/or byte string values the length of the result is equal to the maximum length of the arguments.

Examples

```
1 SELECT
GREATEST(COUNT(*),MIN(salary),AVG(salary), MAX(salary)),
LEAST(COUNT(*),MIN(salary),AVG(salary),MAX(salary))
FROM person;
```

```
Result:
|      50000      |      986      |
```

```
2 SELECT age,salary,GREATEST(age,salary/1000)
FROM person;
```

```
Result:
AGE      SALARY
|      30      |      10800      |      30      |
|      48      |      51000      |      51      |
|      52      |      37000      |      52      |
|      41      |      11000      |      41      |
|      32      |      51000      |      51      |
|      34      |      36000      |      36      |
```

Value Conversion Functions

Conversion to String

Syntax

```
TO_CHAR ( <Value expression> [,<Format string> ] )
```

Description

The result of this function is <Value expression> formatted according to the <Format string> argument. The data type of the <Value expression> is SMALLINT, INTEGER, BIGINT, DECIMAL, or DATE. The data type of the result is CHAR. The data type of the <Value expression> is Numeric data type or DATE data type.

The structure of a <Format string> for a numeric <Value expression> is:

' [S] 9 [9...] [.9 [9...]] '.

The default format string is:

<u>Data Type</u>	<u>Format</u>
Smallint	99999
Int	9999999999
Bigint	99999999999999999999
Real	99999999.99(rounding)
Double	99999999.99(rounding)
Decimal	99999999999999999999.9999999999


The structure of <Format string> options for date and time are:

<u>For Date</u>	<u>For Time</u>
'DD-Mon-YY', default	'HH24'
'DD.Mon.YYYY'	'HH24:MI'
'MM/DD/YY'	'HH:MI:SS'
'MM/DD/YYYY'	'HH24:MI:SS.FF'
'DD.MM.YY'	
'DD.MM.YYYY'	
'DD-Mon-YY', default	'HH24'
'DD.Mon.YYYY'	'HH24:MI'

<u>Item of the Format</u>	<u>Meaning</u>
DD	Day of month (01-31)
MM	Month (01-12)
MON	Abbreviation name of month (3 characters)
YYYY	4-digit year
YY	Last 2 digits of year (00-99)
HH	Hour of day (01-12)
HH24	Hour of day (00-23)
MI	Minute (00-59)

<u>Item of the Format</u>	<u>Meaning</u>
SS	Second (00-59)
FF	Tick , 1/100 of a second (00-99)
FFF	1/1000 of second. (3 digits)

The <Format string> is not case sensitive. 'DD', 'dd', 'Dd', and 'dD' are equivalents.

 Value representation of the items of the Format can not be truncated.

Examples

```
1 TO_CHAR(dat, 'dd-mm-yyyy/hh24:mi:ss')
```

```
Result:
23-10-1997/02:27:38
```

```
2 TO_CHAR(dat, 'Report date: dd/mm/yy')
```

```
Result:
Report date: 23/10/97
```

```
3 TO_CHAR(dat, 'hh24:mi')
```

```
Result: 02.27
```

Remarks

If century digits are omitted, and the number is MORE than 37 the 20th century (digits 19) is implicit. If century digits are omitted, and the number is LESS than 37 the 21st century (digits 20) is implicit.

Conversion to Number

Syntax

```
TO_NUMBER ( <Character value expression> )
```

Description

The result of this function is the numeric value of the <String> argument, or zero if an error occurred during the conversion.

The data type of the result is DOUBLE.

Conversion to Date

Syntax

```
TO_DATE ( <Character value expression> [, <Format string> ] )
```

Description

This function returns the date value obtained from the <Character value> expression argument formatted according to the <Format string>. An empty date is returned if an error occurred during conversion.

The data type of the result is DATE.

See <Format string> argument syntax in section "Conversion to String".

Conversion to Xml

Syntax

XML (<Value>)

Description

This function returns the XML-format of the <Value>.

The data type of the result is VARCHAR(4000).

Example

```
SELECT XML(color) FROM auto;
Result:
| <tr><td>BLACK</td></tr> |
| <tr><td>WHITE</td></tr> |
| <tr><td>BLACK</td></tr> |
| <tr><td>BLACK</td></tr> |
| <tr><td>YELLOW</td></tr> |
| <tr><td>WHITE</td></tr> |
...
```

Conversion to Hexadecimal

Syntax

HEXTORAW (<Character value expression>)

Description

The result of this function is the hexadecimal representation of the <Character value expression> argument where every 2 characters of the argument are represented by byte.

Conversion to Character String

Syntax

RAWTOHEX (<Value expression>)

Description

The result of this function is the character representation of the <Value expression> argument where every byte of the argument is represented by 2 characters.

RAW Conversion Functions

GETBYTE

Syntax

GETBYTE (<Value expression> ,<Byte shift>)
 <Byte shift> ::= unsigned INT

Description

The result of this function is the value of a byte extracted from the <Value expression> argument by the <Byte shift> displacement. <Value expression> may be of any data type.

Possible value of <Byte shift>:

<u>Data type of <Value expression></u>	<u>Possible value of <Byte shift></u>
CHAR(N)	0 - N-1
VARCHAR(N)	0 - N-1
BYTE(N)	0 - N-1
NCHAR(N)	0 - N-1
VARBYTE(N)	0 - N-1
NCHAR VARYING(N)	0 - N-1
DECIMAL(NUMERIC)	0 - 15
BIGINT	0 - 7
INT	0 - 3
SMALLINT	0 - 1
REAL	0 - 3 (according to architecture)
DOUBLE	0 - 7 (according to architecture)
DATE	0 - 15
TRUE(FALSE)	0
BLOB	0-23 (Values from BLOB qualifier)

The data type of the result is I NT.

Example

```
SELECT
GETBYTE('a',0),
GETBYTE('A',0),
GETBYTE(hex('FC0A'),1),
GETBYTE(4567,1),
GETBYTE(TO_CHAR(sysdate,'dd/mm/yyyy'),5)
FROM person WHERE personid=2;
```

```
Result:
| 97 | 65 | 10 | 17 | 47 |
```

GETWORD**Syntax**

```
GETWORD ( <Value expression> ,<Byte shift> )
<Byte shift> ::= unsigned INT
```

Description

The result of this function is the value of a word extracted from the <Value expression> argument by the <Byte shift> displacement. <Value expression> may be of any data type. The data type of the result is INT.

GETLONG**Syntax**

```
GETLONG ( <Value expression> ,<Byte shift> )
<Byte shift>      ::=      unsigned INT
```

Description

The result of this function is the value of a long word extracted from the <Value expression> argument by the <Byte shift> displacement. <Value expression> may be of any data type. The data type of the result is INT.

GETBITS**Syntax**

```
GETBITS ( <Value expression>,<Byte shift>,<Bit shift>,<Bit count> )
<Byte shift>      ::=      unsigned INT
<Bit shift>       ::=      unsigned INT
<Bit count>       ::=      unsigned INT
```

Description

The result of this function is the bit sequence value extracted from <Value expression> and converted to the INT data type. The beginning and the length of the extracted bit sequence are specified by <Byte shift>, <Bit shift> and <Bit count>. The length of the extracted bit sequence cannot exceed 32 bits. <Value expression> may be of any data type.

Example

```
Getbits(hex('01450affcd02'), 2, 3, 8)
```

```
Result:
01010111 (explanation below)
```

Value	01	45	0a	ff	cd	02
Octal representation	00000001	01000101	00001010	11111111	11001101	00000010
Byte shift		2				
Bit shift			012			
Result	00000001	01000101	00001010	11111111	11001101	00000010

GETSTR and GETRAW**Syntax**

```
GETSTR ( <Value expression>,<Byte shift>,<Char count> )
GETRAW ( <Value expression>,<Byte shift>,<Byte count> )
```

<Byte shift>	::=	unsigned INT
<Char count>	::=	unsigned INT
<Byte count>	::=	unsigned INT

Description

The result of these functions is the character or byte substring extracted from the <Value expression> argument. The start of the substring in the string is specified by the <Byte shift> argument. The length of the substring is specified by the <Char count> and <Byte count> arguments.

BLOB Commands and Functions

Linter provides the ability to SEARCH on a blob column based on word criteria. A word in Linter is a contiguous combination of letters and numbers not longer than 31 characters. If a character is not a letter or the number, it is a delimiter. Each BLOB column word includes a property represented by location (in bytes) from the beginning of the blob-value.

COUNTBLOB

This function returns the number of times a specified pattern was found in the specified column.

Syntax

```
COUNTBLOB::= (<Column name>,<Pattern>)
<Pattern>      ::= <string defining search pattern>
```

Description

- 1) <Column name> must be a blob column.
- 2) <Pattern> can contain the wildcard pattern matching characters underscore, _, for a single symbol, and percent, %, to match any number of symbols. See section "LIKE Predicate".
- 3) A non-negative integer is returned.

Examples

```
1  SELECT filename, COUNTBLOB(textblob,'standard %')
   FROM fb ORDER BY filename;
2  SELECT author FROM da WHERE COUNTBLOB
   (da.articletext,'RUSSIAN') >= 10;
```

FINDBLOB

FINDBLOB returns the offset into the BLOB file, referenced by the named BLOB column, of the start of the defined search pattern.

Syntax

```
FINDBLOB::= (<Column name>,<Pattern>,<Iteration>)
<Pattern>      ::= <string defining search pattern>
<Iteration>    ::= <i-th occurrence of the pattern to be found>
```

Description

- 1) <Column name> must be a BLOB column.
- 2) <Pattern> can contain the wildcard pattern matching characters underscore, _, for a single symbol, and percent, %, to match any number of symbols. See section "LIKE Predicate".
- 3) FINDBLOB searches all BLOB files specified by each row of <Column name>.
- 4) If the pattern is found, a non-negative integer is returned specifying the offset into the file of the start of the pattern. Zero is returned if the pattern is not found.

Examples

```

1      SELECT FileName, FINDBLOB(TextBlob,'table', 1)
FROM fb
WHERE FINDBLOB(TextBlob,'table', 2) > 500 ORDER BY filename;

2      SELECT Author, FINDBLOB(ArticleText, 'Russian',
COUNTBLOB(ArticleText, 'Russian') ) AS LastWord
FROM Da
WHERE COUNTBLOB(Da.ArticleText, 'Russian') > 0;
```

GETBLOB

Retrieves a part or segment of BLOB data.

Syntax

```

GETBLOB(<Column name>,<Shift to data>,<Data Size>)
<Shift to data>      ::= <Numeric value expression>
<Data Size>         ::= <Numeric Literal>
```

Description

- 1) BLOB data is considered as contiguous byte array.
- 2) <Shift to data> is unsigned integer which sets the beginning of the data segment.
- 3) <Data size> sets the size of the data segment.

LENBLOB

This function returns the length of BLOB-column.

Syntax

```

LENBLOB (<Column name>)
```

Description

- 1) <Column name> must be a BLOB column.
- 2) Return value is length of the BLOB column in bytes.
- 3) If BLOB column has a NULL value then return value is NULL.
- 4) Data type of the return value is INT.

Example

```
1 SELECT filename, LENBLOB(Rextblob) FROM fb ORDER BY filename;
```

Date Functions

Date functions operate on values of the DATE data type

Date Arithmetic

You can add and subtract numeric constant from the dates.

The constant may be a day, or month or year.

Examples

```
1 SELECT sysdate + TO_DATE('05','yy') FROM person;
```

```
2 UPDATE person SET register=register - TO_DATE('27','dd')
WHERE id=2;
```

DAYNAME

Get the contracted name of the day of the week from a date.

Syntax

```
DAYNAME (<Value expression>)
```

Description

Return the contracted name (3 characters) of the day of the week from <Value expression>. <Value expression> should have DATE data type.

The data type of return value is CHAR.

MONTHNAME

Get the contracted name of the month of the year from a date.

Syntax

```
MONTHNAME (<Value expression>)
```

Description

Return the contracted name (3 characters) of the month of the year from <Value expression>. <Value expression> should have DATE data type.

The data type of return value is CHAR.

TO_GMTIME

Converts a date value at time zone displacement to a UTC (Coordinated Universal Time-formerly Greenwich Mean Time).

Syntax

```
TO_GMTIME (<Date Value expression> [,<time zone>])
```

Description

<time zone> – character literal is Time Zone Abbreviation. For a listing of valid values of the Time Zone Abbreviation go to <http://www.worldtimezone.com/wtz-names/timezonenames.html>.

If <time zone> is omitted then currently set time zone is used.

Example

```
TO_GMTDATE (SYSDATE, 'MDT');
```

TO_LOCALTIME

Converts a date value at UTC to local date.

Syntax

```
TO_LOCALTIME (<Date Value expression> [,<time zone>])
```

Description

<time zone> – character literal is Time Zone Abbreviation. For a listing of valid values of the Time Zone Abbreviation go to <http://www.worldtimezone.com/wtz-names/timezonenames.html>.

Miscellaneous Functions**DBNAME**

Returns the name of opened data base.

Syntax

```
DBNAME ()
```

NULLIFERROR

Ignores error code for data conversion functions.

Syntax

```
NULLIFERROR (<Value expression>)
```

Description

This function handles ERRCONVDATE return code for functions TO_DATE and TO_CHAR_ERRVALRANGE return code for function HEXTORAW.

Return value is NULL if conversion error is occurred or <Value expression> given by the conversion functions.

Example

```
INSERT INTO TEXT (DAT)
```

```
VALUES NULLIFERROR(TO_DATE('08.04.20017', 'YYYY')));
```

DECODE

Provides an abbreviated CASE expression by comparing expression.

Syntax

```
DECODE (<Value expression>, <expression1 >, <result 1 >  
[...<expression N>, <result N> ] <Default value>)
```

Description

Compares <Value expression> to each <expression N> one by one. If <Value expression> is equal to a <expression N> returns <result N>. If no match is found, returns <Default value>.

NVL

Syntax

```
NVL (<Value expression1 >, <Value expression 2>)
```

Description

This function is equal to the following: CASE WHEN <Value expression1> IS NOT NULL THEN <Value expression 1> ELSE <Value expression 2>.

SYS_GUID

This function generates and returns a global identifier.

Syntax

```
SYS_GUID()
```

Description

The return value is a global identifier made up of 16 bytes.

On most platforms, the generated identifier consists of a host identifier and a process or thread identifier of the process or thread invoking the function and a nonrepeating value for that process or thread.

Example

```
SELECT SYS_GUID() FROM AUTO;  
  
|54FA6C5E19733A05E03400400B40DCB1|
```

SQL elements and conceptions

Column Specification

Syntax

```

<Column specification> ::= [<Qualifier>.]<Column name>
<Qualifier> ::= <Table name>
                | <Correlation name>
<Correlation Name> ::= <Identifier>

```

Remarks

- 1) <Qualifier> can reference either to TABLE or VIEW.
- 2) <Column specification> is a reference to a named column. The semantics of such a reference may depend on the context.
- 3) Let C be a <Column Name> and TB be a table associated with <Qualifier> R.
- 4) If the <Column specification> contains a <Qualifier>, <Column specification> must appear within the scope of one or more <Table name>s or <Correlation name>s that is equal to <Qualifier>. If there is more than one such <Table name> or <Correlation name>, the one with the greatest local scope is specified. The associated table must include a column named C. See the definition of scope in section "FROM Clause".
- 5) If the <Column specification> does not contain a <Qualifier>, it should appear within the scope of one or more <Table name>s or <Correlation name>s whose associated tables include a column named C. The most local of such scopes should contain exactly one possible <Table name> or <Correlation name>.
- 6) If the <Column specification> is contained in a table expression, T, and the scope clause of the <Qualifier> is some <SQL statement> or <Table expression> that contains T, the <Column specification> is an outer reference to the table associated with this <Qualifier>.
- 7) The data type of the <Column specification> is the data type of column C included in TB.
- 8) C or R.C usually refers to the value of the column C in a specific row of table T.

Expression

Expressions specify a value.

Syntax

```

<Value expression> ::= <Numeric value expression>
                    | <String value expression>
                    | <Date value expression>
                    | <Condition expression>
<Numeric value expression> ::= <Term>
                    | {<Value expression>{ + | - }<Term>}
<Term> ::= <Factor>
                    | {<Term>{ * | / } <Factor>}
<Factor> ::= [ + | - ]<Primary>

```

<String value expression>	::=	<Term> <String value expression> { +} <Primary>
<Date value expression>	::=	<Primary>
<Condition expression>	::=	See section "Condition"
<Primary>	::=	<Value specification> <Column specification> <Aggregate function specification> <Query> <CAST> <CASE> (<Value expression>)

Remarks

- 1) <Value expression> cannot contain BLOB data type.
- 2) <Value expression> allows using addition and subtraction operation for Data Interval.
- 3) If the data type of a <Primary> is not numeric, <Value expression> can include only '||' and '+' operations.. The data type of the result is the data type of the specified <Primary>.
- 4) If the data type of both operands of a dyadic arithmetic expression is an exact type, the data type of the result is also an exact numeric type, determined as follows:
 - If the data type of either operand is DECIMAL, the data type of the result is DECIMAL.
 - Otherwise, if the data type of either operand is INT, or BIGINT, or SMALLINT, the data type of the result is respectively INT, or BIGINT, or SMALLINT.
- 5) If the data type of either operand of a dyadic arithmetic operator is an approximate numeric type, the data type of the result is also an approximate numeric type. If the data type of either operand is DOUBLE, the data type of the result is DOUBLE too; otherwise, it is REAL.
- 6) If the value of any <Primary> in a <Value expression> is NULL, the result is NULL.
- 7) If a <Value expression> contains only a <Primary>, then the result is the value of the specified <Primary>.
- 8) A monadic plus, +, may be used for human readability, but does not effect its operand. A monadic minus, -, reverses the sign of its operand.
- 9) Attempting to divide by zero generates an error.
- 10) If the data type of a result of an arithmetic operator is an exact numeric type:
 - If the operator is not division and the mathematical result of the operation lacks the exact precision and scale of the result type, an exception is generated.
 - If the operator is division and the mathematical result of the operation has the precision and scale of the result type but loses one or more leading significant digits, as a result of required truncation, an exception is generated.
- 11) Expressions within the parentheses are calculated first, starting at the deepest level of nesting. If the calculation order is not determined by the parentheses, the following priority obtains:
 - Monadic operations;

- Multiplication and Division;
- Addition and Subtraction;

for operations on the same level, from left to right.

- 12) During the process of the numeric literal, if the literal does not contain decimal point and exponent sign, attempt to convert it to type INTEGER is made. If conversion to INTEGER produces an error, attempt to convert it to type BIGINT is performed. If the conversion to INTEGER and BIGINT failed, and literal does not contain the exponent sign, attempt to convert it to DECIMAL is made. In case of failure, literal is converted to type DOUBLE.
- 13) If <Value expression> applies to the table row, then the specified column of the table is referenced by current column value in that row.
- 14) An exceptions which is generated during some impropriety operations can be avoid using following:
 - Include a naught check in division operator and set up particular result value.

Example:

```
CREATE TABLE tab1 (i1 dec, i2 dec not null);
INSERT INTO tab1 VALUES (1,2);
INSERT INTO tab1 VALUES (1,0);
SELECT * FROM tab1;
```

Result:

```
| 1.0 | 2.0 |
| 1.0 | 0.0 |
```

```
SELECT CASE i2 WHEN 0 THEN 'Attempting to divide by zero' ELSE
CAST i1/i2 AS CHAR END FROM tab1;
```

Result:

```
| 0.5 |
| Attempting to divide by zero |
```

- Add some additional predicates to manage NULL values (CHECK or NOT NULL during set up of column preferences).

Example:

```
CHECK (c1 *c2 is NOT NUUL and c1 *c2>5000);
```

Examples

Value expression as <Value specification>:

```
SELECT COUNT(*) FROM person WHERE name = 'Adkinson';
SELECT * FROM person WHERE name = user;
SELECT rownum, COUNT(*) FROM person GROUP BY name;
```

Value expression as <Column specification>:

```
SELECT DISTINCT make FROM auto;
```

Value expression as <Aggregate function specification>:

```
SELECT DEFAULT(make) FROM auto;
```

Value expression as <Value expression>:

```
SELECT (DEFAULT(make)) FROM auto;
SELECT (rownum), (COUNT(*)) FROM person GROUP BY name;
```

Value expression as <Subquery>:

```
SELECT model FROM auto WHERE auto.personid =  
(SELECT personid FROM person WHERE auto.personid =  
Person.personid and name = 'Anderson');
```

Cast

CAST specifies an explicit data type conversion.

Syntax

```
CAST [(|<Expression> AS <Data type>|)]  
| <String value expression>
```

Remarks

- 1) <Expression> must be scalar (not a subquery) and must be of type numeric, character, date or can be NULL value. <Data type> may include specifications of length, precision, and scale.
- 2) Any numeric expression can be explicitly converted into any other numeric data type. If the result of conversion, with possible rounding, does not fall into the range of values of the result type, an error condition is raised.
- 3) A string expression can be converted explicitly into any numeric data type. Both the leading and trailing blanks of the string expression are truncated. The remaining characters are converted to numeric value according to SQL literal syntax. If the conversion is impossible, an error condition is generated.
- 4) Explicit conversion of a string into a string with a different length is possible. If the length of the result is greater than the length of the argument, the result is padded with blanks. If the length of the result is less than the length of the argument, then the argument is truncated. Error conditions are not returned for these conversions.
- 5) If the explicitly specified length of a string is insufficient to contain the converted value, the rightmost part of the converted value is truncated, and no error condition is returned.
- 6) The NULL-value is recognized as a declared type.
- 7) Any numeric expression can be converted into a string. If the length of a string type is not specified, it is assumed to be equal to:
 - SMALLINT: char(6)
 - INTEGER: char(11)
 - REAL or DOUBLE: char(20)
 - DECIMAL: char(32)
- 8) There are implicit conversions which are supported in Linter:
 - NULL to any data type.

For literals:

- CHAR->VARCHAR;
- INTEGER->SMALLINT;
- INTEGER->BIGINT;
- DOUBLE->REAL;

- DECIMAL->DOUBLE;
- DECIMAL->REAL.

Example

```
SELECT CAST PersonID AS SMALLINT FROM Auto;
```

Case**Syntax**

CASE is a conditional statement. It has two forms:

```
1 CASE <Value expression1 >
    {WHEN <Value expression2(n)> THEN <Action1(n)>}...
    [ ELSE <Action2>]
END
```

```
2 CASE
    { WHEN <Condition(n)> THEN <Action1 (n)>} ...
    ELSE <Action2>
END
```

Remarks

Form 1:

- 1) <Value expression1 > and <Value expression2> must be of the same data type. If they are of CHAR or BYTE type, and have different lengths, the returned length will be the longer of the two.
- 2) Let R(n) be the result of each comparison of <Value expression1 > & <Value expression2(n)>.
- 3) If any R(n) is TRUE, <Action1(n)> will be taken. If every R(n) is FALSE, <Action2> will be taken. If ELSE is omitted and every R(n) is FALSE, NULL is returned.
- 4) It must be possible to compare <Value expression1> and each <Value expression2(n)>.

Examples

```
SELECT CASE make WHEN 'ford' THEN 1 WHEN 'HONDA' THEN 2 ELSE 3
END FROM AUTO;
```

```
SELECT DISTINCT make, model FROM AUTO WHERE make = CASE
cylinders WHEN 8 THEN UPPER('ford') ELSE UPPER('Chrysler') END;
```

Form 2:

- 1) If any <Condition(n)> is true, <Action1(n)> is taken. If no <Condition(n)> is true, <Action2> is taken. If the option is omitted, and no <Condition(n)> is true, NULL is returned.

Examples

The following two queries return the same answer:

```
SELECT CASE Make WHEN 'FORD' THEN 1
```

```
WHEN 'FERRARI' THEN 2 ELSE 3 END FROM Auto;
```

```
SELECT CASE WHEN Make = 'FORD' THEN 1
WHEN Make = 'FERRARI' THEN 2 ELSE 3 END FROM Auto;
```

Condition

A condition returns a result that is true, false, or unknown, depending on the result of application of Boolean operators to specified predicates.

Syntax

```
<Condition> ::= <Boolean term>
              |<Condition> OR <Boolean term>
<Boolean term> ::= <Boolean factor>
                  |<Boolean term> AND <Boolean factor>
<Boolean factor> ::= [NOT] <Boolean primary>
                    IS [NOT] {TRUE | FALSE}
<Boolean primary> ::= <Predicate> | (<Condition>)
```

Remarks

- 1) The result is derived by applying Boolean operators to the results derived from each <Boolean primary>. If Boolean operators are not specified, the result of the <Condition> is the result of the specified <Boolean primary>.
- 2) The expressions within parentheses are calculated first. If the order of calculations is not determined by the parentheses, the priority of the operators are: NOT, then AND, then OR. The priority of identical operators is from left to right.
- 3) NOT true is false, NOT false is true, and NOT unknown is unknown. AND and OR results are defined in the following tables.

AND	True	False	Unknown
True	TRUE	FALSE	UNKNOWN
False	FALSE	FALSE	FALSE
Unknown	NULL	FALSE	UNKNOWN

OR	True	False	Unknown
True	TRUE	TRUE	TRUE
False	TRUE	FALSE	UNKNOWN
Unknown	TRUE	UNKNOWN	UNKNOWN

Predicates

Predicates specify a condition that can be evaluated to give a Boolean value.

Syntax

```
<Predicate> ::= { <Comparison predicate>
                  | <BETWEEN predicate>
                  | <IN predicate>
```

```

| <LIKE predicate>
| <EXISTS_Predicate>
| <NULL predicate>
| <Quantified predicate>}

```

Remarks

A predicate is applied to a table row.

Examples

```

1 Salary BETWEEN 100000 AND 200000
2 Color IN ( 'YELLOW, 'RED' )
3 Make LIKE 'F_R of %'
4 Name IS NOT NULL
5 EXISTS ( SELECT * FROM Auto WHERE Color = 'CYAN' )

```

Comparison Predicates

These predicates specify a comparison of two values.

Syntax

```

<Comparison Predicate> ::= <Value expression 1>
                        <Comparison operator>
                        {<Value expression 2>
                        |(<Subquery> ) }

<Comparison operator> ::= = | <> | < | > | <= | >=

```

Remarks

- 1) The data types of <Value Expression 1> and <Subquery> or <Value Expression 2> must be comparable.
- 2) Let X be the value of <Value Expression 1> and Y be the value of <Subquery> or <Value Expression 2>. The result of X <Comparison operator> Y may be regarded as being evaluated using the following sequence:
 - If <Subquery> returns more than one value, an exception is generated.
 - If either X or Y is NULL or the result of the <Subquery> is empty, the result is undefined.
 - If neither X nor Y is neither NULL nor empty, the result is either TRUE or FALSE; TRUE if the comparison condition is met, otherwise FALSE.
- 3) In contrast to usage in this <Comparison predicate>, a NULL value is considered equal to any other NULL value in GROUP BY, ORDER BY and DISTINCT clauses.

Example

```

CREATE TABLE T (I int);
INSERT INTO T VALUES (NULL);
INSERT INTO T VALUES (NULL);
SELECT I FROM T;

```

```

Result:
| NULL |
| NULL |

```

```
SELECT DISTINCT I FROM T;
```

Result:

```
| NULL |
```

- 4) <Value expression> evaluated in BOOLEAN can be convert automatically to <Comparison predicate>. For example: the query "SELECT * FROM T WHERE B;" is identical to the query "SELECT * FROM T WHERE B=TRUE;".
- 5) Numeric values are compared with respect to their algebraic values.
- 6) CHAR string comparison is described in section "CHAR, CHARACTER".
- 7) BYTE string comparison is described in section "BYTE".
- 8) VARCHAR string comparison is described in section "CHAR VARYING, CHARACTER VARYING, VARCHAR".
- 9) VARBYTE string comparison is described in section "BYTE VARYING, VARBYTE".
- 10) NCHAR string comparison is described in section "NATIONAL CHAR, NARIONAL CHARACTER, NCHAR" and section "NATIONAL CHAR VARYING, NATIONAL CHARACTER VARYING, NCHAR VARYING".
- 11) BLOB and EXTFILE columns are not supported in comparison predicate.

BETWEEN Predicate

Specifies a range comparison.

Syntax

```
<BETWEEN predicate> ::= <Value expression > [ NOT ]
                        BETWEEN <bound> AND <bound>
<bound> ::= {<Value expression> |<Select query> }
```

Remarks

- 1) The data types of all <Value expression> must be comparable.
- 2) <Select query> must return only single value;
- 3) Let X, Y and Z be the first, second and third <Value expressions>, respectively.
- 4) The expression X BETWEEN Y AND Z is equivalent to X >= Y AND X <= Z.
- 5) The expression X NOT BETWEEN Y AND Z is equivalent to NOT (X BETWEEN Y AND Z)

Examples

- 1 Select count (*) from AUTO where cylinders between 6 and 8;
- 2 Select count (*) from AUTO where cylinders>=6
and cylinders <=8;
- 3 Select count (*) from AUTO where cylinders not between 6
and 8;
- 4 Select count (*) from AUTO where not cylinders between 6
and 8;

IN Predicate

The IN predicate specifies a quantified comparison.

Syntax

```
<IN predicate>      ::= <Value expression>
                       [ NOT ] IN {(<Subquery>) | (<Value list> ) }
                       | (<Value expression>[,...]) [NOT]IN (<Subquery>)
<Value list>        ::= <Value specification> [,...]
```

Remarks

- 1) Data types of the <Value expression> and the <Subquery> must be comparable.
- 2) Data types of all <Value specification> in the <Value list> must be implicitly convertible to the data types of the <Value expression>.
- 3) Number of columns of the result row of the <Subquery> must agree with number of <Value expression>.
- 4) Because comparison of two NULL values results in FALSE the rows containing NULL values are excluded from the result.
- 5) Let X be the first <Value expression>, S be the result of <Subquery> or set of values specified by <Value list>.
- 6) X IN S is equivalent to X = ANY S.
- 7) X NOT IN S is equivalent to NOT (X IN S). The following queries are equivalent:

```
Select name, count(*) from person where age not in
(30,40,50,60) group by name;
```

```
Select name, count(*) from person where not age in
(30,40,50,60) group by name;
```

Example

```
1 SELECT Name FROM Person / * Auto Owners * /
WHERE PersonID IN ( SELECT PersonID FROM Auto );
```

```
2 CREATE TABLE TAB1 (I int autoinc, C char(10), DT date);
INSERT INTO TAB1 (C,DT) VALUES ('abcdef','01/01/2003');
INSERT INTO TAB1 (C,DT) VALUES ('bcdefa','01/01/2003');
INSERT INTO TAB1 (C,DT) VALUES ('bcdefab',NULL);
SELECT * FROM TAB1;
```

Result:

```
| 1| abcdef| 01/01/2003:00:00:00.00|
```

```
| 2| bcdefa| 02/01/2003:00:00:00.00|
```

```
| 3| cdefab | NULL |
```

```
3 SELECT * FROM TAB1 WHERE (I,C,DT) IN
(SELECT I,C,DT FROM TAB1 WHERE C like '_cde%');
```

Result:

```
| 2 | bcdefa | 02/01/2003:00:00:00.00 |
```

LIKE Predicate

LIKE enables a pattern-match comparison.

Syntax

```

<LIKE predicate>          ::= <Column specification> [ NOT ] LIKE
                           <Pattern> [ESCAPE<Escape character>]

<Pattern>                 ::= <String literal> |<Character expression>
<Escape character>       ::= <String literal >
                           (single-character defining the escape character)

```

Remarks

- 1) The data type of the <Column specification> must be CHAR, VARCHAR, NCHAR, or NCHAR VARYING.
- 2) Let X be the value of <Column specification>, Y be the <Pattern>, and E be the <Escape character>.
- 3) Cases:
 - If, in a 2 character substring of Y beginning with E, the second character is another E, the underscore, or the percent, an exception is raised.
 - Other 2 character substrings of Y beginning with E represent a single occurrence of the second character of the substring.
 - Each substring of length 1 that is the underscore character matches an arbitrary single character.
 - Each substring of length 1 that is the percent character matches an arbitrary string.
 - Each substring of length 1 that is neither the underscore character nor the percent character represents the character that it contains.
 - If no <Escape character> is specified:
 - Each underscore character in Y matches an arbitrary single character in X.
 - Each percent character matches an arbitrary string in X.
 - Each character that is neither the underscore character nor the percent character represents itself.
- 4) Default <Escape character> is '\'.
- 5) If X is NULL, X LIKE Y is undefined. Otherwise, X LIKE Y is either true or false.
- 6) X LIKE Y is true if and only if X matches the pattern
- 7) X NOT LIKE Y is equivalent to NOT (X LIKE Y).

Examples

```

1  SELECT DISTINCT make FROM auto WHERE UPPER(make)
   LIKE ('%MOTOR%');

```

```

Result:
|AMERICAN MOTORS|

```

```
|GENERAL MOTORS |
2 SELECT DISTINCT bodytype FROM auto WHERE
UPPER(bodytype) LIKE ('_E%');
```

Result:

```
|SEDAN |
|SEDAN HARDTOP |
```

```
3 SELECT DISTINCT make,bodytype FROM auto WHERE bodytype LIKE
('SE' || 'DAN' || %);
```

```
4 SELECT DISTINCT make,bodytype FROM auto WHERE bodytype LIKE
('SE' || 'DAN' ||?);
```

```
5 SELECT COUNT(1) FROM "Corporation"
WHERE UPPER("Corporation_Name") LIKE 'SONY/_%' escape '/';
```

EXISTS Predicate

EXISTS tests for a non-empty set.

Syntax

```
<EXISTS_Predicate> ::= EXISTS (<Subquery>)
```

Remarks

- 1) The result of EXISTS <Subquery> is either true or false. It is true if and only if the result of <Subquery> is not empty.

Examples

```
SELECT Name FROM Person / * Auto Owners * /
WHERE EXISTS (SELECT * FROM Auto
FROM PersonID = Person. PersonID);
```

```
SELECT min(a1 .id+1) as answer FROM tab as a1
WHERE NOT EXIST
(SELECT 1 FROM tab as a2 WHERE a1 .id+1 = a2.id);
```

NULL Predicate

NULL tests for a null value.

Syntax

```
<NULL predicate> ::= <Column specification> IS [NOT] NULL
```

Remarks

- 1) Let X be a value derived from <Column specification>.
- 2) X IS NULL can be either true or false.
- 3) X IS NULL is true, if and only if X is a NULL value.
- 4) X IS NOT NULL is equivalent to NOT (X IS NULL).

Example

```
SELECT * FROM Person WHERE Age IS NOT NULL;
```

QUANTIFIED Predicate

Specifies a quantified comparison.

Syntax

```
<Quantified predicate>      ::=  <Value expression>
                               <Comparison operator>
                               <Quantifier> (<Subquery>)
<Quantifier>                ::=  ALL | {SOME | ANY}
```


Remarks


- 1) Data types of <Value expression> and <Subquery> must be comparable.
- 2) Let X be the <Value expression> result, S be the <Subquery> result, and R be each row returned by S.
- 3) The <Quantified predicate> result is derived by applying the implied <Comparison predicate> to X and then applying the <Comparison_operator> to every R returned in S.
- 4) Cases:
 - If S is empty or if the implied <Comparison predicate> is true for every R in S, then X <Comparison operator> ALL S is true.
 - If the implied <Comparison predicate> is false for at least one R in S, then X <Comparison operator> ALL S is false.
 - If the implied <Comparison predicate> is true for at least one R in S , then X <Comparison operator> SOME | ANY S is true.
 - If S is empty or if the implied <Comparison predicate> is false for every R in S, then X <Comparison operator> SOME | ANY S is false.
 - If <Quantified predicate> is neither true nor false, it is undefined.

Examples

```
1 SELECT Name FROM Person / * With Min Salary * /
WHERE Salary <= ALL (SELECT Salary FROM Person);
```

```
2 SELECT Name FROM Person /* Auto Owners */
WHERE PersonID = ANY ( SELECT PersonID FROM Auto );
```

 Quantified predicates can be easily transformed one into another using negation: e.g., ALL to ANY and back.

 Quantified predicate queries can usually be better formulated using the EXISTS (see section "EXISTS Predicate") or IN (see section "IN Predicate") predicates or the aggregate functions (see section "Aggregate functions").

Oracle Style Join Operator (+)

A JOIN is a query that combines rows from two or more tables, or views.

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. For compatibility with Oracle Linter also supports the condition is called an

Oracle style OUTER JOIN counterparts of these keywords, (+)= and =(+). For ANSI syntax of OUTER JOIN see section "JOINS".

Syntax

<Oracle style OUTER JOIN>	::=	LEFT [OUTER] JOIN RIGHT [OUTER] JOIN FULL [OUTER] JOIN
LEFT [OUTER] JOIN	::=	<Column specification>(+) = <Value expression>
RIGHT [OUTER] JOIN	::=	<Value expression> = <Column specification>(+) =
FULL [OUTER] JOIN	::=	<Column specification>(+) = <Column specification>(+) =

Remarks

- 1) See section "JOINS".
- 2) You can not specify the (+) operator in a query block that also contains ANSI JOIN syntax (see section "JOINS").

Table Expressions

Table expressions return a table or a grouped table.

Syntax

<Table expression>	::=	<FROM clause> [<WHERE clause>] [<GROUP BY clause>] [<HAVING clause>] [<ORDER BY clause>];
--------------------	-----	--

Remarks

- 1) If all optional clauses are omitted, the result of the <Table expression> is the result of the <FROM clause>. Otherwise, each specified clause is applied to the result of the previously specified clause, and the final result of <Table expression> is the result of the application of the last specified clause.
- 2) The result of the table expression is the table obtained, in which the i-th column derives the declaration of the i-th column of the table(s) specified by FROM.

Examples

- 1 SELECT COUNT(*) FROM AUTO, PERSON;
- 2 SELECT COUNT(*) FROM AUTO, PERSON
WHERE AUTO.personid = PERSON.personid;

FROM

Specifies a table derived from one or more (up to 16) named tables.

Syntax

<FROM clause>	::=	FROM<Table reference> [, ...]
<Table reference>	::=	<Table name>[[AS]<Correlation name>]

		< Derived table> [AS]<Correlation name>
		<Joined table>
<Derived table>	::=	(<Query expression>)
<Joined table>	::=	See section "JOINS"

Remarks

- 1) The <Table name> specified in a <Table reference> is exposed in the FROM clause that contains it if, and only if, the <Table Reference> does not specify a <Correlation name>.
- 2) Any <Table name> exposed in a FROM clause cannot be the same as any other <Table name> exposed in the same FROM clause.
- 3) A <Correlation name> specified in a FROM clause cannot be the same as any other <Correlation name> or <Table name> specified in the same FROM clause.
- 4) The scope of <Correlation name>s and exposed <Table name>s in a subquery of a FROM clause is limited to the subquery (a <Subquery>, <Query specification>, or SELECT operator) that contains the FROM clause's <Table expression>. This scope does not extend to the FROM clause itself.
- 5) Cases:
 - If a FROM clause consists of a single <Table reference>, the returned table has the same description as that specified in the <Table reference>.
 - If a FROM clause contains more than one <Table reference>, the description of the returned table is a concatenation of the descriptions of the tables specified by those <Table reference> and in the same order as they occurred in the FROM clause.
 - Specification of the <Correlation name>s and exposed <Table name>s in the <Table reference> specifies the use of those <Correlation name>s and <Table name>s in the returned table, which will have the identifier specified by <Table name> or <Derived table> in that <Table reference>.

Examples

```

1 SELECT * FROM Person, Auto, Finance;

2     SELECT      *      FROM      Auto      A,      Person      Prs      WHERE
A.PersonID=Prs.PersonID;

3 SELECT * FROM Auto AS A1, Auto AS A2;

4 SELECT * FROM (SELECT * FROM Person P, Auto A
WHERE P.PersonID=A.PersonID ) AS Pa
(SELECT * FROM Person P, Finance F
WHERE P.PersonID=F.PersonID ) AS Pf;

```

WHERE

WHERE returns a table derived by application of a <Condition> to the result of a preceding FROM clause.

Syntax

```
<WHERE clause> ::= WHERE <Condition expression>
```

Remarks

Let T be the result of a precedent FROM clause.

- 1) Each <Column specification> directly contained in the <Condition expression> must unambiguously reference a column in T or be an outer reference.

Example

```
SELECT B.name, A.model FROM auto A, person B
WHERE A.personid = B.personid AND
EXIST (SELECT * FROM finance C WHERE
C.personid = A.personid AND crditlim 1000);
```

- 2) If a <Value expression> directly contained in the <Condition expression> is an aggregate function, the WHERE clause must be contained in a HAVING clause, and the argument to WHERE must be an outer reference.

Example

```
SELECT college FROM finance GROUP BY college HAVING
COUNT(college) > 10;
```

- 3) The <Value expression> directly contained in the <Condition expression> must not include reference a column which is a result of aggregate function.
- 4) The <Condition expression> is applied to each row of T. The result of the WHERE clause is a table of those rows of T for which the result of the <Condition expression> is true.
- 5) Each <Subquery> in the <Condition> is executed for each row of T and the results are used in the application of the <Condition> to the current row of T. If any executed <Subquery> contains an outer reference to a column of T, then the reference is to the value of that column in the current row of T.

Examples

```
/* Select names receiving average salary */
/* from a group with the same name */
SELECT FirstName FROM Person P GROUP BY FirstName
HAVING FirstName
IN (SELECT FirstName FROM Person WHERE Salary = AVG(P.Salary));
```

GROUP BY

The results of the immediately prior clause are placed in a derived table in the sequence specified by GROUP BY.

Syntax

<GROUP BY clause>	::=	GROUP BY <group specification> [{,<group specification>}...]
<group specification>	::=	{<Value expression> <Column specification>}

Remarks

Let T be the result of a precedent FROM clause.

- 1) Let T be the result of the precedent WHERE clause or the result of the precedent FROM clause, if no WHERE clause is specified.

- 2) Each <Column specification> in a GROUP BY clause must unambiguously reference a column of T. A column referenced in a GROUP BY clause is a grouping column. A <Value expression> specified in a GROUP BY clause is a grouping expression.
- 3) The result of the GROUP BY clause is a division of T into a set of groups. The set is the minimum number of groups such that, for each grouping column for each group of more than one row, no value in that grouping column is unique.
- 4) When a <Condition> or <Value expression> is applied to a group, the reference to a grouping column is the reference to the common value for every row of that group.
- 5) Select clause can use a grouping expression.
- 6) Columns, which have data type BLOB or EXTFILE, cannot be grouping columns.
- 7) <Condition> of WHERE clause related to GROUP BY can not contain CASE or CAST expressions.

Examples

```
1 SELECT FirstName FROM Person P GROUP BY
   FirstName
   HAVING FirstName
   IN (SELECT FirstName FROM Person WHERE Salary = AVG
      (P.Salary));
```

```
2 SELECT ROUND(weight/1000) as 'Weight', model
   FROM auto GROUP BY ROUND(weight/1000), model;
```

```
3 SELECT model, COUNT(*) FROM auto WHERE year = 70 GROUP BY
   model;
```

HAVING

HAVING returns a grouped table derived by elimination of groups that do not meet the <Condition>.

Syntax

```
<Having clause> ::= HAVING <Condition>
```

Remarks

Let T be the result of the previous FROM clause and any previous WHERE or GROUP BY clauses.

- 1) Each <Column specification> contained directly in the <Condition> should unambiguously reference a grouping column of T or be an outer reference.
- 2) Each <Column specification> contained in a <Subquery> in the <Condition> that references a column of T, must be a grouping column of T or must be an argument of an aggregate function.
- 3) If there is no GROUP BY clause, T consists of a single group.
- 4) The <Condition> is applied to each group of T. The HAVING clause returns a grouped table of those groups of T for which the result of the <Condition> is true.
- 5) When the <Condition> is applied to a group of T, that group is the argument of each aggregate function contained directly in the <Condition> unless the argument to the aggregate function is an outer reference.

- 6) Each subquery in the <Condition> is executed for each group of T. The result is then used in the application of the <Condition> to the given group of T. If a <Subquery> contains an outer reference to a column of T, then the reference is to the value of that column in the given group of T.

Examples

```
1 SELECT DISTINCT make FROM auto
GROUP BY make
HAVING MAX(year) >
SELECT MIN(auto.year) FROM auto, person
WHERE person.personid = auto.personid AND
LENGTH(auto.make) = LENGTH(person.name));
```

```
2 SELECT make FROM auto GROUP BY make
HAVING MAX(year+1900) <
(SELECT TO_NUMBER(TO_CHAR(sysdate, 'YYYY')));
```

```
3 SELECT make, cylinders, color FROM auto
WHERE color IN ('BLACK', 'WHITE', 'GREEN')
GROUP BY make, cylinders, color HAVING MAX(cylinders) > 6;
```

```
4 SELECT make, year+1900 FROM auto GROUP BY make, year+1900
HAVING MAX(year+1900) BETWEEN 1969 AND 1972;
```

ORDER BY

The sorted order in which the SELECT query result is presented is controlled by the ORDER BY clause.

Syntax

```
<ORDER BY clause> ::= ORDER BY <Order sequence>[,...]

<Order sequence> ::= {Unsigned INT
|<Column name>
| <Value expression>}
[ASC | DESC]
```

Remarks

Let T be the result of a SELECT query.

- 1) The sequence of T's rows is controlled by <Order sequence>.
- 2) Each <Column specification> must unambiguously specify a column of T.

Examples

```
1) SELECT * FROM auto ORDER BY model;
2) SELECT DISTINCT model, make FROM auto ORDER BY model DESC;
3) SELECT DISTINCT model, make, cylinders FROM auto
ORDER BY model ASC, cylinders DESC;
```

- 3) Each Unsigned INT must be greater than zero and less than or equal to the number of columns in T.
- 4) A named column may be specified by an Unsigned INT or by a <Column name>. An unnamed column must be specified by an Unsigned INT.

Example

```
SELECT DISTINCT model, make, TO_CHAR(weight) || 'pounds'
"Weight", year+1900 AS "Year" FROM auto
WHERE rownum = 1 ORDER BY model, 3, 4;
```

- 5) Unsigned INT specifies the i-th column of T <Column name> specifies a named column of T.
- 6) ASC , the default, specifies a column's ascending order. DESC specifies descending order.
- 7) The sequence in <Order sequence> determines the sequence of sorting columns. The 1st column specified in <Order sequence> is sorted first. The 2nd column is sorted next, etc.
- 8) Columns, which have BLOB or EXTFILE data type, can not be used in GROUP BY clause.

Joins

A JOIN is a query that combines rows from two or more tables, or views. A join is performed whenever multiple tables appear in the query's <FROM clause>.

Syntax

```
<Joined table> ::= <Table reference> <Join type> JOIN
<Table reference>[ON <Join condition>]
[WAIT | NOWAIT];
<Join type> ::= CROSS
| [INNER]
| LEFT [OUTER]
| RIGHT [OUTER]
| FULL [OUTER]
| UNION
<Table reference> ::= [<User name>.]<Table name> [[AS]<Alias>]
| (<Subquery>) [AS] <Alias>
| <Joined table>
| (<Table reference>)
```

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. For compatibility with Oracle Linter also supports the condition is called an Oracle style OUTER JOIN counterparts of these keywords, (+)= and =(+) (see section "ORACLE STYLE JOIN OPERATOR (+)").

Remarks

- 1) If parentheses are not used in the join expression, it will be executed from left to right.
- 2) The scope of the each joined table is the subquery that generated it.
- 3) CROSS JOIN (equivalent to JOIN) is a query that combines rows from two or more tables.

Examples

```
1 All three statements are equivalent:
2 SELECT * FROM A CROSS JOIN B;
3 SELECT * FROM A Join B;
4 SELECT * FROM A,B;
```

- 4) You can not use a ON <Join condition> phrase with CROSS JOIN, but WHERE clause can be used.

- 5) An INNER JOIN (also known as simple join) is a join of two or more tables that returns only those rows that satisfy the join condition.
- 6) A “Tab1 LEFT [OUTER] JOIN Tab2” returns all rows of Tab1 table and some rows of Tab2 table that satisfy the join condition. For all rows in Tab1 table that have no matching rows in Tab2 table, Linter returns null for any <Select list> containing columns from Tab2 table. Alternative Oracle style syntax see section "ORACLE STYLE JOIN OPERATOR (+)".

Examples

Let be a two tables:

tab1 (i int) and tab2 (i int), which have following rows:

```
tab1 tab2
1     2
2     4
3     5
4     7 ,
then
```

```
1 SELECT tab1.i, tab2.i FROM tab1 LEFT JOIN tab2 ON tab1.i =
tab2.i;
```

Alternative Oracle style syntax:

```
SELECT tab1.i,tab2.i FROM tab1, tab2 WHERE tab1.i(+) = tab2.i;
```

Result:

```
1      Null
2      2
3      Null
4      4
```

```
2      SELECT tab1.i, tab2.i FROM tab2 LEFT JOIN tab1 ON tab1.i =
tab2.i;
```

Alternative Oracle style syntax:

```
SELECT tab1.i,tab2.i FROM tab1, tab2 WHERE tab2.i(+) = tab1.i;
```

Result:

```
2      2
4      4
null   5
null   7
```

- 7) A “Tab1 RIGHT [OUTER] JOIN Tab2” returns all rows of Tab2 table and some rows of Tab1 table that satisfy the join condition. For all rows in Tab2 table that have no matching rows in Tab1 table, Linter returns null for any <Select list> containing columns from Tab1 table. Alternative Oracle style syntax see section "ORACLE STYLE JOIN OPERATOR (+)".

Example

```
SELECT tab1.i , tab2.i FROM tab1 RIGHT JOIN tab2 ON tab1.i =
tab2.i;
```

Alternative Oracle style syntax:

```
SELECT tab1.i,tab2.i FROM tab1, tab2 WHERE tab1.i = tab2.i(+);
```

```
Result:
2          2
4          4
Null       5
Null       7
```

- 8) A “Tab1 FULL [OUTER] JOIN Tab2” returns all rows from both tables, extended with nulls if they do not satisfy the join condition. Alternative Oracle style syntax see section "ORACLE STYLE JOIN OPERATOR (+)".

Example

```
SELECT tab1 .i , tab2.i FROM tab1 FULL JOIN tab2 ON tab1 .i =
tab2.i;
```

Alternative Oracle style syntax:

```
SELECT tab1 .i,tab2.i FROM tab1, tab2
WHERE tab1 .i(+) = tab1 .i(+);
```

```
Result:
2          2
4          4
null       5
null       7
1          null
3          null
```

- 9) A “Tab1 UNION JOIN Tab2” returns all rows of Tab1 table and some rows of Tab2 table that have no matching rows in Tab1 table.
- 10) You cannot specify the (+) operator in a query block that also contains ANSI JOIN syntax.

Query Specification

A table to be derived from a <Table expression> is described by a query specification.

Syntax

```
<Query specification> ::= <Query Expression> [<ORDER BY clause>]
                        [<FETCH FIRST clause>]
                        [<FOR UPDATE clause>]
                        [<FOR BROWSE clause>]
                        [WAIT|NOWAIT];
```

Query Expression

Query expression combines the results of two component queries into a single result. Queries containing a set of queries are called compound queries.

Syntax

```
<Query expression> ::= <Query expression>
                      { UNION | INTERSECT | EXCEPT }
                      [ALL |DISTICT] < Query expression >
                      (<Query expression>)
                      | [(|<Subquery>|)];
```

All set operators have equal precedence. If a SQL statement contains multiple set operators, Linter evaluates them from the left to right if no parentheses explicitly specify another order.

<u>Operator</u>	<u>Returns</u>
UNION	all rows selected by either <Query term>.
UNION ALL	all rows selected by either <Query term>, including all duplicates.
EXCEPT ALL	all rows selected by first <Query term> but not the second.
EXCEPT DISTINCT	all distinct rows selected by first <Query term> but not the second.
INTERSECT DISTINCT	all distinct rows selected by both <Query term>s.
INTERSECT ALL	all rows selected by both <Query term>s.

Remarks

Let T be the table specified by the <Query expression>.

- 1) If the UNION keyword is not specified, and the <Query specification> is updatable, then T is an updatable table.

Cases:

- If UNION is not specified, T is the result of the <Query specification>.
 - For each two <Query term>s used in a <Query expression>, the data types of the corresponding columns must be identical, but column names may be unique.
- 2) The ORDER BY clause must be contained only in the last SELECT clause, and must specify the ordering of columns by their numbers only.
 - 3) Linter's SQL allows named columns to be returned from a <Query expression>. This allows the use of the returned, joined table in a subquery in a FROM clause.
 - 4) A column returned as the result of any <Query expression> operators is a named column if, and only if, all columns of source tables from which the given column is received have the same name. This name, then, becomes the name of the returned column.

Examples

```
//start first query specification
SELECT Name, 'Green Auto' FROM Person,Auto
WHERE Color='GREEN' AND Person.PersonID=Auto.PersonID
//end first query specification
UNION
//start second query specification
SELECT Name, 'Black Auto' FROM Person,Auto
WHERE Color='BLACK' AND Person.PersonID=Auto.PersonID
//end second query specification
ORDER BY 1;
```

In this example, the UNION of two SELECTs has one named column (the first column named Name) and one unnamed column, the second.

Fetch First

Explicitly retrieve only the first desired rows of a query.

Syntax

```
<FETCH FIRST clause> ::= FETCH FIRST [<number>]
```

Remarks

- 1) <Number> is unsigned integer.
- 2) If <number> is omitted then FETCH FIRST returns the first single row.
- 3) If the <number> is greater than the number of result rows then all rows of a query returns.

Examples

```
1 SELECT rowed, make, color FROM auto FETCH FIRST 2;
```

Result:

```
| 1 | FORD      | BLACK      | | 2 | ALPINE | WHITE      |
```

```
2 SELECT rowid, make, color FROM auto WHERE rowed > 2
   FETCH FIRST 3;
```

Result:

```
| 3 | AMERICAN MOTORS | BROWN |
| 4 | MASERATI         | BLACK |
| 5 | CHRYSLER         | WHITE |
```

FOR UPDATE clause

This clause specifies whether updates are allowed through a cursor opened on the query.

Syntax

```
<FOR UPDATE clause> ::= FOR UPDATE [ OF <Column name> [,...]]
```

Remarks

- 1) FOR UPDATE clause lets you lock the selected rows so that other users cannot lock or update the rows until COMMIT or ROLLBACK.
- 2) FOR UPDATE clause can be specified only in a top-level SELECT statement (not in subqueries) and FROM clause's <Table expression> must contain only one <Table name>.
- 3) If the number of result rows is greater than 1000 then entire table will be locked.
- 4) Of <Column name> clause has no effect on. This clause is used only for support of ANSI/ISO SQL -89, SQL-92 standards.

FOR BROWSE

This clause specifies whether updates are allowed through a cursor opened on the query.

Syntax

```
<FOR BROWSE clause> ::= FOR BROWSE
```

Remarks

- 1) FOR BROWSE clause obtains lock only of currently processed row.

WAIT/NOWAIT

WAIT and NOWAIT clauses let you appoint how to proceed if the SELECT statement attempts to lock a row that is locked by another user.

WAIT is specified then current transaction will wait until other transaction will finish and table will be unlocked. But if NOWAIT is specified than transaction won't wait and error occur.

Subquery

A subquery describes a table obtained by the union of one or more <Query specification>s. (query nested within another SQL statement is called a subquery).

Syntax

<Subquery>	::=	{SELECT [ALL DISTINCT] <Select List> [<FROM clause> [<WHERE clause> [<GROUP BY clause> [<HAVING clause>]] <Hierarchical query>;
<Select List>	::=	{<Derived column>[, ...] } *
<Derived column>	::=	<Value expression>[[AS]<Column name> ROWNUM ROWTIME ROWID
<Hierarchical query>	::=	See section "Hierarchical query"

Remarks

Let R be the result of one of <Table expression>s and C be a <Column name>.

- 1) The user's privileges for any table named in <FROM clause> must include SELECT.
- 2) The size of the table specified by a <Subquery> is equal to the number of elements in the <Select list>.
- 3) The <Select list>'s "*" argument generates a <Subquery> sequence in which each <Column specification> references a column of R, and each column of R is referenced exactly once.
- 4) Each column reference in each of the <Value expression>s must unambiguously reference a column of R.
- 5) If R is a grouped table, then each <Column specification> in each <Value expression> must reference a grouping column or must be within an aggregate function. If R is not a grouped table, but some <Value expression> contains an aggregate function, each <Column specification> in each <Value expression> must be within an aggregate function.

- 6) The data type of each column of the result table is the same as the data type of the <Value expression> from which the column was derived.

Cases:

- If the i-th <Derived column> in the <Select list> specifies an AS clause that contains a <Column name> C, then the i-th column of the result is a named column with the name C.
 - If the i-th <Derived column> in the <Select list> does not have an AS clause and the <Value expression> of that column is a single <Column specification>, the i-th column of the result is a named column with a <Column name> equal to the name of the column referenced by <Column specification>.
 - Otherwise, the i-th column of the result is unnamed.
- 7) A <Subquery> is either updatable or not updatable. A <Subquery> is updatable if, and only if, all the following conditions are satisfied:
- the DISTINCT keyword is not specified;
 - each <Value expression> in the <Select list> is a single <Column specification>;
 - the <FROM clause> contains exactly one <Table reference>;
 - the table referenced by this <Table reference> is updateable;
 - the <WHERE clause> does not include any <Subquery>;
 - the <Table expression> includes neither a GROUP BY clause nor a HAVING clause.
- 8) If R is not a grouped table and the <Select list> contains an aggregate function, R is an argument to each such function, and the result of the <Query specification> is a table consisting of one row. The i-th value of the row is the value specified by the i-th <Value expression>.
- 9) If R is not a grouped table, and <Select list> does not contain an aggregate function, each <Value expression> is applied to each row of R generating a table, T, of M rows, where M is the number of rows in R. If DISTINCT is not specified, the result of <Query specification> is T. If DISTINCT is specified, T is derived from R by the elimination of any duplicate rows.
- 10) If R is a grouped table that has no groups, and any <Value expression> in <Select list> is a single <Column specification>, the result of the <Subquery> is an empty table. If R is a grouped table and it has no groups, and every <Value expression> in the <Select list> is an aggregate function, the result of the <Subquery> is a table consisting of one row. The value of the i-th column of this row is the result of the i-th aggregate function.
- 11) If R is a grouped table that has one or more groups, each <Value expression> is applied to each group of R. The resulting table, T, has M rows, where M is number of groups in R. When a <Value expression> is applied to a given group of R, that group is an argument to each function in the <Value expression>. If the DISTINCT keyword is not specified, the result of the <Subquery> is T. If DISTINCT is specified, the result of the <Subquery> is a table derived from T by eliminating any duplicate rows.
- 12) A row is a duplicate of another row if, and only if, all <Value expression> have the same values for both rows.
- 13) The ROWNUM pseudo-column presents the current row number in the result set.

- 14) If <Table expression> contains reference to table that is locked by other transaction and if WAIT is specified then current transaction will wait until other transaction will finish and table will be unlocked. But if NOWAIT is specified than transaction won't wait and error occurs.

Examples

See examples in the section "QUERY SPECIFICATION".

Hierarchical Query

CONNECT BY clause identifies the relationship between higher and lower level rows of the hierarchy.

Syntax

```
<hierarchical query> ::= SELECT <Select list>
                        FROM <Table name>[ [AS] <alias>]
                        [START WITH <Condition>]
                        CONNECT BY <Relational condition>
                        [WHERE clause];
<relational condition> ::= {PRIOR <Expression>
                            <Comparison operator> <Expression>}
                        | {<Expression><Comparison operator>
                            PRIOR <Expression>}
```

Remarks

- 1) START WITH clause specifies a condition that identifies the root rows of a hierarchical query. If START WITH clause is omitted then all rows in the table will be used as root rows.
- 2) The condition in the CONNECT BY clause defines the relations between higher and lower level rows. The conditions that define the rows of the previous level must contain a PRIOR clause.
- 3) The <FROM clause> can contain only one table. By default, the columns returned are the same columns as in <Table name> plus one column. This additional column is named LEVEL and is of type INTEGER. It contains the row level number in the hierarchy. LEVEL returns the value 1 for a root level, 2 for a higher level of a root level, 3 for a higher level, and so on.
- 4) PRIOR evaluates the <Relational condition> for the higher level row of the current level row.
- 5) You may use START WITH, CONNECT BY, and <WHERE clauses> in any order.
- 6) Column LEVEL can not be used in START WITH and CONNECT BY clauses.

Examples

```
SELECT name, position, supervisor
FROM employees START WITH position = 'manager'
CONNECT BY PRIOR supervisor = name;
```

Data Definition Language (DDL)

Create Table

Defines a base table.

Syntax

<Table Definition>	::=	CREATE TABLE [OR REPLACE] <Table name> [<Code Page> (<Table element list> [<Parameter>[, ...]] [AS <subquery>];
<Code Page>	::=	CHARACTER SET < Code page name> DEFAULT
<Table element list>	::=	<Column definition> [{,<Column definition> <Constraint definition>} ...]
<Column definition>	::=	<Column name> <Column type> [Code Page] [DEFAULT <Default value> [<Column constraint>[,...]
<Column type>	::=	See section "Data Types"
<Default value>	::=	SYSDATE NULL USER <Literal>
<Column constraint>	::=	[<Reference definition> NOT NULL AUTOINC [INITIAL (<initial value>) AUTOINC RANGE (<Range description> AUTOROWID UNIQUE PRIMARY KEY CHECK (<Condition>]
<Range description>	::=	<Range>[,...]
<Range>	::=	<beginning range> :<ending range>
<References definition>	::=	{REFERENCES <Referenced table name> [(<Referenced column name>[,...])] [<Action specification>] }
<Action specification>	::=	[ON UPDATE {CASCADE SET NULL SET DEFAULT {NO ACTION RESTRICT} }] [ON DELETE {CASCADE SET NULL SET DEFAULT {NO ACTION RESTRICT} }]
<Constraint definition>	::=	PRIMARY KEY(<Column name>[, ...]) FOREIGN KEY (<Column name>[, ...])

		UNIQUE (<Column name>[,...])
		CHECK (<Condition>)[,...]
<Parameter>	::=	<Table parameter> <File parameter>
<Table parameter>	::=	{MAXROWID
		MAXROW
		PCTFILL
		BLOBPCT} <Unsigned INT>
<File parameter>	::=	{INDEXFILES
		DATAFILES
		BLOBFILES}
		<Number files>
		[(<File description>[, ...])]
<File description>	::=	[<Device name>] [<File size>]
<Number files>	::=	Unsigned INT
<Device name>	::=	<String literal>
<File size>	::=	Unsigned INT

Remarks

- 1) The <Table definition> defines a base table.
- 2) <Table name> must be distinct from the name of any existing table or view created by the same user.
- 3) If the table name isn't started with the double quote (") character, the first character must be a letter.
- 4) The table name can contain any characters if surrounded by double quotes.
- 5) <Table name> is limited to 66 characters.
- 6) <Column definition> describes each column in the <Table definition>. The description of the i-th column is specified by the i-th <Column definition>.
- 7) <Table definition > must include at least one <Column definition>.
- 8) The column must be described before using the <Constraint definition>.
- 9) Each <Column name> must have a name that is unique among the <Column name>s in all <Column description>s in this <Table definition>.
- 10) The maximum number of columns in a table is 250.
- 11) [DEFAULT <Default value>] and [<Column constraint>[,...]] clauses can be used in any order.

Example

The following statements are equivalent.

```
CREATE TABLE TAB1 (i int DEFAULT null UNIQUE);
CREATE TABLE TAB1 (i int UNIQUE DEFAULT null);
```

- 12) If NOT NULL is not specified and a <Default definition> clause is not used, DEFAULT NULL is implicit.
- 13) If <Column definition> contains a DEFAULT clause, neither the <Column constraint> not <Constraint definition> can set PRIMARY KEY on one column as a constraint. But <Column constraint> or <Constraint definition> can set composite Primary key (index on two or more columns) as a constraint.
- 14) NOT NULL and DEFAULT NULL can be used in one <Column definition>.

- 15) <Default value>s can be used only for columns.
- 16) The AUTOINC column must be of type INTEGER, SMALLINT, or BIGINT.
- 17) AUTOINC and PRIMARY KEY are compatible in <Column constraint> and <Constraint definition>.

Examples

```
CREATE TABLE test1 (I INT PRIMARY KEY AUTOINC);
CREATE TABLE test1 (I INT AUTOINC, PRIMARY KEY (i));
```

- 18) If default value is entered into an AUTOINC column then this default value will be the starting value for AUTOINC column otherwise it will be 1. There are 4 AUTOINC columns allowed to create within one table.

Example

```
CREATE TABLE tab1 (I INT AUTOINC DEFAULT 10, si SMALLINT
AUTOINC, bi BIGINT AUTOINC DEFAULT -1000);
INSERT INTO tab1 DEFAULT VALUES;
INSERT INTO tab1 DEFAULT VALUES;
INSERT INTO tab1 DEFAULT VALUES;
INSERT INTO tab1 DEFAULT VALUES;
SELECT * FROM tab1;
```

Result:

10	1	-1000
11	2	-999
12	3	-998
13	4	-997

- 19) The new entered Range is not checked. It must be done by administrator of Data Base which creates this range.
- 20) If a default value is entered into an AUTOINC column with a RANGE modifier, it should be the lowest among those values that are higher than the last entered AUTOINC value and fall within one of the indicated ranges.
- 21) If an expressly indicated value is entered into an AUTOINC column with a RANGE modifier, it should be lower than the last entered AUTOINC value only if it is within one of the specified ranges. Otherwise, any value may be entered.
- 22) If Range modifier is specified then INITIAL cannot be set and vice versa.
- 23) Default value entered into an AUTOINC column with a RANGE modifier cannot be higher than <range end> + 1.
- 24) The range beginning should always be LESS than the end of this range, while the beginning of the next range should be less than the end of the previous range.

Example

```
CREATE TABLE TEST (I INT AUTOINC RANGE(1:1000,
5000:10000, 70000: 100000));
```

- 25) A column having the AUTOWID modifier must be of INTEGER, SMALLINT, or BIGINT type;
- 26) Each table may have only one AUTOWID column;
- 27) An AUTOWID column may not have the AUTOINC modifier.

- 28) If records in a table containing an AUTOROWID column are updated, updating values in the AUTOROWID column is forbidden.
- 29) If a table contains an AUTOROWID column, records are entered into this table in the following way:
- if the value for the AUTOROWID column has not been set, a value for each record should be entered into this column; each value should equal the ROWID of the corresponding record;
 - if the value for the AUTOROWID column has been set, the record should be entered into the table, the ROWID being equal to the indicated value. If the specified ROWID is occupied by another record, or a forbidden ROWID value is indicated, an error is detected.
- 30) The value of AUTOROWID column must be a positive integer not greater than MAXROWID. PRESS TABLE is forbidden.
- 31) AUTOROWID can not be used with CHECK constraint.
- 32) The CHECK constrain specifies a condition that each row in the table must satisfy.
- 33) CHECK constraint on column can not reference to any other columns in the table.

Example

```
CREATE TABLE t (i INT CHECK (i>0));
```

- 34) The <condition> of a CHECK constraint on table can refer to any columns in the table, but it cannot refer to columns of other tables.

Example:

```
CREATE TABLE t (i INT, j INT, CHECK (i<j));
```

- 35) There are no restrictions on the number of CHECK constraints, but the size of the constrain condition cannot exceed 4 KByte.
- 36) CHECK constraint conditions cannot contain the subqueries, aggregate functions, CASE, and CAST clauses.
- 37) The <Referenced table name> and <Referenced column name> in the <References definition> declare the referenced column and table.

Examples:

```
CREATE TABLE ref_sub (id INT PRIMARY KEY, Name CHAR(10),
Id_ref INT REFERENCES par_tab(id_key));
```

```
CREATE TABLE tab1 (I INT PRIMARY KEY, c CHAR(10),
ref_i INT REFERENCES tab1(i));
```

- 38) It is not necessary to specify the column name in the REFERENCES construction, if the reference is to the primary key of the table, whether the key includes one or more columns.

Examples

```
CREATE TABLE tab1 (i INT, c1 CHAR(10), c2 VARCHAR(15),
PRIMARY KEY (I,c2));
```

```
CREATE TABLE tab2 (i INT REFERENCES tab1,c2 VARCHAR(20));
```

- 39) The specified action determines how the row will be executed for updates or deletes:

- CASCADE executes the update or delete on the referenced column and row;
- SET NULL sets the referred column to NULL;
- SET DEFAULT sets the referred column to its default value;
- NO ACTION | RESTRICT cancels this SQL statement.

Examples

1 References on column:

```
CREATE TABLE tab (id INT PRIMARY KEY, name CHAR(10),
id_ref1 INT REFERENCES tab(id)
ON DELETE CASCADE, id_ref2 INT REFERENCES parent_tab(id) ON
DELETE SET NULL);
```

2 References on table:

```
CREATE TABLE tab (id INT PRIMARY KEY, name CHAR(10), id_ref1
INT,
FOREING KEY (id_ref1) REFERENCES parent_tab(id) ON DELETE
CASCADE);
```

- 40) By default, the <Action specification> is NO ACTION | RESTRICT.
- 41) RESTRICT is a synonym for NO ACTION.
- 42) PRIMARY KEY clause in <Constrain definition> designs a column or combination of columns as the primary key. The primary key is made up of a combination of column is called composite primary key. For a composite primary key no two rows in the table can have the same combination of values in the key columns.
- 43) There is only one Primary Key allowed on table.
- 44) UNIQUE clause in <Constrain definition> designs a column or combination of columns as the unique key. The unique key is made up of a combination of column is called composite unique key. Any key columns in composite unique key can contain null-values. For a composite unique key no two rows in the table can have the same combination of values in the key columns.
- 45) A MAXROWID parameter specifies the maximum ROWID of the table to be created. MAXROWID must be a positive integer. The default is 1024.
- 46) A MAXROW parameter specifies the maximum number of rows that can be simultaneously stored in the table. This value must be greater than zero and cannot exceed MAXROWID. The MAXROW parameter is significant to other table parameters such as file sizes. By default, Linter calculates these sizes using MAXROW. By default, MAXROW is equal to MAXROWID.
- 47) The PCTFILL parameter defines the percent value of data compression. It is a positive integer not greater than 100; the default is 100%.
- 48) BLOBPCT is the percent of a BLOB page to be filled. This is an integer number from 1 to 100. The default is 50.
- 49) The value of <Number files> cannot be greater than 63.
- 50) If the <File parameter>s are omitted, the table created will have one data file and one index file.
- 51) <File size> specifies the number of 4096-byte pages in the file. This page size is independent of hardware page size. It cannot be less than 2. If <File size> is not specified, Linter calculates file size according to other relevant parameters and default settings.

- 52) By default Maximum size of row is 4K, but you can increase size up to 65,535 using command ALTER DATABASE SET RECORD SIZE LIMIT. (only for version 6.0)
- 53) <Device name> specifies an operating system environment variable defining a logical name for the device or path-to-a-directory in which the table file is to be created. <Device name> cannot have more than 4 characters. The following steps are taken to resolve <Device name> when not specified in the <Device description>:
 - if the \$\$\$DEVICE table is in the database and the <Device name> is defined there, the file will be created on that device;
 - if <Device name> is not in the database, Linter will try to get it from the environment variable;
 - by default, Linter use SY00 as the environment variable for <Device name>;
 - if SY00 is not defined, Linter will create the table file in the current directory.
- 54) AS <subquery> clauses specifies a subquery to determine the contents of the table. The rows returned by the subquery are inserted into the table upon its creation.

Examples

```
1 CREATE TABLE auto (PersonID INT NOT NULL, Make CHAR(20)
  DEFAULT', Model CHAR(20)) MAXROWID 4096 MAXROW 1000 PCTFILL 70;

2 CREATE TABLE IntegerTable (IntegerColumn INTEGER NOT NULL)
  PCTFILL 50 INDEXFILES 1(4) DATAFILES 1('DB', 2);
```

Default Column Values

Specifies the default value for a column.

Syntax

```
<Default value> ::= {<Literal>
                    | USER
                    | SYSDATE
                    | NULL}
```

Remarks

- 1) The data type of <Default value> is the data type of the column specified in the <Column description> containing it.

Cases:

- The data type of the <Default value> is the data type of the column specified in the <Column description> containing the default value.
- If SYSDATE is specified, the data type of the <Default value> is DATE.
- If NULL is specified, the <Column description> cannot include a NOT NULL clause.

- 2) When a row is inserted into a table but a value for a column is not specified, that column is filled by the <Default value>.

Cases:

- If the <Column description> does not contain a DEFAULT definition or if the <Column description> explicitly or implicitly defines DEFAULT NULL, the column is initialized with a null value by default.

- If the <Column description> specifies DEFAULT <Literal>, the column is initialized with the value of the literal.
- If the <Column description> specifies DEFAULT USER, the column is initialized with the name of the current user padded with blanks on the right to a length of 66.
- If <Column description> specifies SYSDATE, the column is initialized with the current date value.

Drop Table

Deletes a base table.

Syntax

```
<Drop table> ::= DROP TABLE <Table name> [CASCADE];
```

Remarks

- 1) If <Table name> contains <Owner name>, the <Owner name> must be the same as the name of the current user.
- 2) You cannot drop the table that is currently being used by another users.
- 3) If there is "ON DELETE CASCADE" integrity constraint that refers to primary and unique keys in the dropped table then you must specify CASCADE clause in order to delete such table.
- 4) Only the owner of the table can delete the table. An administrator of the DB can delete all objects, including table, of some user using command: DROP USER <user name> CASCADE.
- 5) System tables can not be deleted.

Examples

```
DROP TABLE Auto;
DROP TABLE System.Auto;
```

Create View

Defines a view.

Syntax

```
<View Definition> ::= CREATE [OR REPLACE]
VIEW <View name> [ ( <Column list> ) ]
AS <Query expression>
[ WITH CHECK OPTION ];

<View name> ::= <Table name>
<Column list> ::= <Column name> [, ... ]
```

Remarks

- 1) If <View name> contains the <Owner name>, that <Owner name> must be the same as the name of the current user.
- 2) <View name> must be unique among the tables and views created by the same user.

- 3) If <Query expression> is updateable, the view is an updateable view. Otherwise, the view is read only.
- 4) If <Query expression> is updateable, the view is an updateable view. Otherwise, the view is read only.
- 5) All <Column name>s in <Column list> must be unique within the list.
- 6) The number of names in <Column list> must be equal to the number of table columns in the <Query expression>.
- 7) If WITH CHECK OPTION is specified, the view must be updateable.
- 8) Let V be the result of the <Query expression>. The view is available to Linter only when V is returned.
- 9) If V can be updated, let T be the table specified in the FROM clause of <Query expression>.
- 10) Then for each row, v, from V, there is a corresponding row in T from which v is derived. For each column in V, there is a corresponding column in T. Insertion, deletion, or update of a row of V is the same operation on the corresponding row of T.

Cases:

- If WITH CHECK OPTION is specified and the <Query expression> contains a WHERE clause, the information inserted or updated in the view will be checked for conformity with this WHERE clause.
- If WITH CHECK OPTION is not specified, the view does not check inserted or updated values.

Examples

```

1 CREATE VIEW OwnerAuto AS
  SELECT Name,FirstNam,SerialNo,Make,Model
  FROM Person P,Auto A WHERE P.PersonID =A.PersonID;
2 /* Updatable view */
CREATE VIEW Black_Auto(PersID,Make,Model,SerNo)
AS SELECT PersID,Make,Model,SerNo
FROM Auto WHERE Color ='BLACK';

```

Drop View

Deletes a view.

Syntax

```
<Drop view> ::= DROP VIEW <View name>;
```

Remarks

- 1) If <View name> contains the name of the owner, that name must be the same as the name of the current user.
- 2) Only the owner of the view can delete it. Administrator of DB can delete all objects, including VIEW, of some user using command: DROP USER <user name> CASCADE.
- 3) DROP TABLE can also be used to delete a view. This is particularly if you don't know the type of object you need to delete.

Grant

Grants one or more privileges to one or more users on table or view.

Syntax

```

<Privileges_Definition> ::= GRANT{<Privileges>
                           ON <Table_Name> | <View_Name>
                           TO <Users>}
                           {<User_Category> TO <User_List>
                           IDENTIFIED BY <Password_List>};
<Privileges>             ::= ALL [PRIVILEGES] | <Action_List>
<Users>                  ::= PUBLIC | <User_List>
<User_Category>         ::= CONNECT | RESOURCE | DBA
<Password_List>         ::= <Password> [ , ... ]
<Action_List>           ::= <Action>[ , ... ]
<User_List>             ::= <User_Name> [ , ... ]
<Action>                 ::= SELECT
                           | INSERT
                           | DELETE
                           | UPDATE
                           | ALTER
                           | INDEX
<Password>              ::= <STRING_Literal>

```

Remarks

- 1) ALL includes every privilege listed in <Action>.
- 2) If PUBLIC is specified instead of the list of the users, the specified rights on the specified table are granted to all existing and future database users.
- 3) The second form of the GRANT operator is used to define new users and their privileges.
- 4) <User list> and <Password list> may have a different numbers of elements. The <User list> cannot be shorter than the <Password list>. The correspondence of user and password is implemented from the left to the right. The users remaining without passwords are assigned the empty password, 18 blanks.
- 5) <Password> string can contain up to 18 characters.
- 6) Privileges can be granted only by the owner of the specified table or view. If <Owner name> is included in <Table name>, <Owner name> must be the same as the name of the owner of the table.
- 7) Only a DBA category user can create a new user or change a user's privilege level.

Examples

```

1 GRANT SELECT ON Auto TO PUBLIC;
2 GRANT RESOURCE TO User_1, User_2 IDENTIFIED BY 'Hi_Life';

```

Grant Execute

Grants one or more privileges to one or more users on execution of stored procedure.

Syntax

```

<Grant execute privileges> ::= GRANT EXECUTE [AS OWNER]

```

```

<User name> ::= ON [<Owner name>.]< Procedure name>
              TO <User name>[,...];
              <Identifier>
    
```

Remarks

- 1) Privileges can be granted only by the owner of the specified procedure.
- 2) If AS OWNER is specified then any user can execute procedure.

Revoke

Revokes privileges for one or more users on table or view.

Syntax

```

<Revoke privileges> ::= REVOKE {<Privileges>
                          ON<Table name>|<View name>
                          FROM <Users>}
                          | {<User level> FROM <User list>};
<Privileges>          ::= ALL [PRIVILEGES] | <Action List>
<Action List>        ::= <Action>[ , ... ]
<Action>             ::= SELECT
                          | INSERT
                          | DELETE
                          | UPDATE
                          | ALTER
                          | INDEX
<Users>              ::= PUBLIC | <User_List>
<User List>          ::= <User_Name> [ , ... ]
<User level>         ::= CONNECT | RESOURCE | DBA
    
```

Remarks

- 1) The syntax rules are the same as for GRANT.
- 2) Only the owner of the table can revoke privileges on it. If <Table name> includes <Owner name>, that <Owner name> must be the same as the name of the current user.
- 3) If PUBLIC clause specified then all privileges of each user, who has been granted the privileges through PUBLIC, is revoked. However the privileges are not revoked from users who have been granted the privileges directly.
- 4) Only a user with DBA access can revoke or lower a user's privilege level.
- 5) CONNECT level access privileges are included in the RESOURCE level, RESOURCE level privileges are included in the DBA level.
- 6) When a user's privilege level is revoked, he automatically has the next lower privilege level.
- 7) If a user's CONNECT level privilege is revoked, all his access rights are revoked.

Examples

- 1 REVOKE SELECT ON Auto FROM PUBLIC;
- 2 REVOKE RESOURCE FROM User_1, User_2;

Revoke Execute

Revokes privileges to one or more users on execution of stored procedure.

Syntax

```
<Revoke execute privileges> ::= REVOKE EXECUTE [AS OWNER]
                                ON [<Owner name>.]< Procedure name>
                                TO <User name>[,...];
<User name> ::= <Identifier>
```

Remarks

- 1) Privileges can be revoked only by the owner of the specified procedure.
- 2) <Owner name> must be current User name.

Create Index

Creates an index for a table column.

An index on two or more columns is called a composite index.

Syntax

```
<Index definition> ::= CREATE [OR REPLACE]
                    INDEX <Column name> ON <Table name>
                    [INDEXFILE <File number>] [BY APPEND];

<Named index > ::= CREATE [OR REPLACE] INDEX
                 <Index name> ON <Table name>
                 (Column list) [ INDEXFILE <File number>];

<Index name> ::= <Identifier>
<Column list> ::= <Column name>[,...]
<File number> ::= Unsigned INT
```

Remarks

- 1) The user must have the INDEX privilege on <Table name>.
- 2) <Column name> must exist in <Table name>.
- 3) <Table name> must reference a base table, not a view.
- 4) If an index file number is not specified, file number 1 is the default.
- 5) An index is created/deleted automatically when a key is created/deleted (excepting AUTOROWID columns), including non-empty tables.
- 6) For compound indexes implicitly generated when compound keys are announced, the name is generated as "Index#number#", where "number" is the record ROWID for the index/key in the \$\$\$ATTRI table.
- 7) No more than 31 columns can form a composite index.
- 8) The size of the index cannot exceed 1024 Byte.
- 9) Indexes can be created for table with no rows.

- 10) Composite indexes cannot be created for system tables.
- 11) CREATE INDEX for system table is time consuming and the server will not process request referencing the same table while the statement is being processed.
- 12) INDEX always created automatically on PRIMARY KEY, FOREIGN KEY, and UNIQUE columns.
- 13) If BY APPEND ids specified then index creation is executed without data sorting.

Example

```
CREATE INDEX PersonID ON Auto INDEXFILE 2;
```

Drop Index

Deletes an index.

Syntax

```
<Drop index clause> ::= DROP INDEX <Column name>
                        ON <Table name>;

<Drop named index>   DROP INDEX <Index name>
                        ON <Table name>;
```

Remarks

- 1) An index can be deleted only by the owner of the table or by a user who has the INDEX privilege on <Table name>.
- 2) <Column name> must exist in <Table name>.
- 3) <Table name> must reference a base table, not a view.

Example

```
DROP INDEX PersonID ON Auto;
```

Alter Table

Adds a column to a table; adds or deletes an index, data, or blob file.

Syntax

```
<Alter statement> ::= ALTER TABLE
                    <Table Name> |<View Name>
                    {<Alter Table Clause>
                    | ALTER [COLUMN] <Column Name>
                    <Alter Column Clause>};

<Alter Table Clause> ::= ADD (<Column Definition>,...)
                    | ADD COLUMN <Column Definition>
                    | ADD CHECK (<Comparison Predicate>)
                    | ADD {INDEXFILE DATAFILE BLOBFILE}
                    [ <File Declarator>]
                    | ADD FOREIGN KEY (<Column List>)
                    REFERENCES<Foreign Table Name>
                    [ (<Foreign Column List>)] <Action specification>
                    | ADD PRIMARY KEY (<Column List>)
                    | ADD UNIQUE (<Column List>)
```

```

| DROP CHECK
| DROP {INDEXFILE|DATAFILE|BLOBFILE}
| DATAFILE
| BLOBFILE}
| DROP FOREIGN KEY (<Column List>)
[ WITHOUT INDEX]
| DROP PRIMARY KEY
[ WITHOUT INDEX]
| DROP UNIQUE (<Column List>)
[ WITHOUT INDEX]
| MODIFY {INDEXFILE
| DATAFILE
| BLOBFILE}
<File Number><File Declarator
| RENAME TO <New Table Name>
| SET LEVEL (<RAL>,<WAL>)
| SET COLUMN <Column Name>
LEVEL (<RAL>,<WAL>)
<Alter Column Clause> ::= ADD RANGE
(<Beginning Range>:<Ending Range>[,...])
| ADD CHECK (<Comparison Predicate>)
| CHARACTER SET {<Code name>
| DEFAULT}
| DROP CHECK
| DROP DEFAULT
| DROP DEFAULT FILTER
| DROP ROOT
| {DISABLE|ENABLE} NULL
| RENAME TO <New Column Name>
| SET DEFAULT <Default Value>
| SET DEFAULT FILTER <Filter Name>
| SET ROOT <Root Path Value>
| SIZE <New size>
<Action specification> ::= [ ON UPDATE
{ CASCADE
| SET NULL
| SET DEFAULT
| {NO ACTION | RESTRICT} } ]
[ ON DELETE
{ CASCADE
| SET NULL
| SET DEFAULT
| {NO ACTION | RESTRICT} } ]
<Column List> ::= <Column Name>[,...]
<File Declarator> ::= ([<Device String>][<Initial Size>])
<Default Value> ::= {SYSDATE|USER|<Literal>}

```

Remarks

- 1) For most options <Table name> must be a base table, not be a view. If <Table Name> is a view, then only RENAME TO <Table Name> and RENAME TO <Column Name> options are allowed.
- 2) <Column Name> must be unique within <Table Name>.
- 3) The syntax of <Column definition> is described in CREATE TABLE.
- 4) A new column is added at the end of the table's column list.
- 5) The new entered Range is not checked for ADD RANGE option.
- 6) There may be a maximum of 63 files for each <File type>. A DROP clause specifies the deletion of the highest numbered file. It is impossible to delete file number one.
- 7) The DISABLE NULL construction sets the NOT NULL column restriction, while ENABLE NULL cancels the restriction.
- 8) If the option WITHOUT INDEX is specified, the index for the dropped key is not deleted, even if there are no more keys using this index.
- 9) The operator ALTER TABLE ALTER COLUMN SIZE is forbidden in case if the columns are included in compound indexes or defined as a PRIMARY KEY. This clause can be used only for CHAR, NCHAR, VARCHAR, NVARCHAR, BYTE, VARBYTE and only without DEFAULT clause.
- 10) In case if you want to set new size for indexed column, or column included in composite index, or PRIMARY KEY column you should delete index first. Then you can change the size of column and create index again.
- 11) The size of the column may be only increased.
- 12) The operator ADD [COLUMN] <Column Definition> is forbidden for AUTOROWID, BLOB, and EXTFILE columns.
- 13) SET ROOT|DROP ROOT are intended for set/drop default directory for EXTFILE columns. This option cannot be used on database created before v.5.8.
- 14) SET DEFAULT FILTER|DROP DEFAULT FILTER are intended for set/drop default filter for previously assigned column. This option cannot be used on database created before v. 5.8
- 15) <Foreign Column List> in ADD FOREIGN KEY clause can be omitted if Foreign Table contains only one PRIMARY KEY column.

Example

```
CREATE TABLE tab1 (i1 INT);
CREATE TABLE tab2 (i2 INT PRIMARY KEY);
ALTER TABLE tab1 ADD FOREIGN KEY (i1) REFERENCES tab2;
```

- 16) DROP CHECK on table/column deletes all check conditions of table/column.

Examples

```
1 ALTER TABLE Auto ADD DATAFILE ('SY11' 25);
```

```
2 ALTER TABLE REFSUB1A ADD FOREIGN KEY (ID_REF) REFERENCES
REF_MAIN(ID) ON DELETE NO ACTION; /* ON DELETE: §10.4.11.1
*/
```

```
3 ALTER TABLE REFSUB1B ADD FOREIGN KEY (ID_REF1) REFERENCES
REF_SUB1 A(ID) ON DELETE CASCADE;

4 ALTER TABLE REFSUB1A ADD FOREIGN KEY (ID_REF2) REFERENCES
REF_SUB1B (ID) ON DELETE SET NULL;

5 ALTER TABLE REFMAIN ADD PRIMARY KEY (ID);

6 ALTER TABLE REFSUB1A DROP FOREIGN KEY (ID_REF);

7 ALTER TABLE REFMAIN DROP PRIMARY KEY;
```

Rebuild

To facilitate renewal (as accurate as possible) of a table with defective data structure. Rebuild is used when TEST TABLE command issues error.

Syntax

```
<Rebuild table> ::= REBUILD TABLE [<User Name>.]< Table Name>
[WITH INDEXES];
```

Remarks

- 1) <Table name> must reference a base table, not a view.
- 2) <Table name> can not reference a system table.
- 3) To rebuild table you must be granted ALTER privilege to the table and RESOURCE level access.
- 4) The command REBUILD TABLE WITH INDEXES is forbidden for system tables.
- 5) The construction WITH INDEXES indicates the necessity of rebuilding all the indexes in the table (simple and compound).
- 6) Information contained in DATA file is enough to restore all data structures.
- 7) The index files are rebuilt in accordance with the new ROWID reference table.
- 8) Recoverable table is locked during the rebuilding process.

Press

Compression (compaction) of a table after numerous record deletions.

Syntax

```
<Table Compression> ::= PRESS TABLE <Table Name>
[ WAIT | NOWAIT ];
```

Remarks

- 1) <Table Name> should refer to a base table.
- 2) To compress table you must be granted ALTER privilege to the table and RESOURCE level access.
- 3) PRESS TABLE is prohibited for table with AUTOROWID column.
- 4) If WAIT / NOWAIT is not specified then WAIT is set by default.

- 5) In case the table contains no rows PRESS TABLE sets the initial value for AUTOINC INT as defined INITIAL and for AUTOINC BIGINT as 1.

Test Table

Checks the physical structure of the table.

Syntax

```

<Table Testing>          ::=  TEST TABLE [<User Name>.]<Table Name>
                           <Testing action>[ WAIT | NOWAIT ];
<Testing action>        ::=  {COLUMN <Column name> INDEX}
                           | {INDEX <Index Name>}
                           | {INDEX <Column name>}
                           | (<Structure element>[,...])
<Structure element>     ::=  DESCRIPTION
                           | BITMAP
                           | DATA
                           | INDEX
                           | BLOB
                           | INTEGRITY
  
```

Remarks

- 1) <Table Name> should refer to a base table.
- 2) To compress table you must be granted ALTER privilege to the table and RESOURCE level access.
- 3) If WAIT / NOWAIT is not specified then WAIT is set by default.
- 4) During the testing process the testing table is locked in SHARE mode.
- 5) If no structure element is specified then all elements of table will be tested.

Truncate Table

Rapidly removes all rows from a table.

Syntax


```

<truncate table>        ::=  TRUNCATE TABLE
                           [<User Name>.]<Table Name>
                           [ WAIT | NOWAIT ];
  
```

Remarks

- 1) <Table name> should refer to a base table.
- 2) <Table name> should not refer to a system table.
- 3) The right to table truncation belongs to its owner, or to any other user having the table structure modification privilege (ALTER) RESOURCE database access rights level.
- 4) If the WAIT (NOWAIT) modifier is not set, WAIT is used by default.
- 5) The truncated table should not have any references from non-empty tables of the database.
- 6) This command is a DDL operation, i.e. COMMIT is executed before processing it. It is impossible to roll back the command that has already started.

- 7) The entire table is blocked for writing up to the end of the operation.
- 8) All files of the table (data, index, BLOB data) are deleted.
- 9) New table files will be created in minimum quantities (1 data file; 1 index file, if there are no index description references from several index files; 1 BLOB file, if there are BLOB columns, or if long records are possible) and in minimum sizes: one bitmap page (4 Kbytes), and the data file, index file and BLOB file one page each).

 The TRUNCATE TABLE command is not equivalent to DROP TABLE and CREATE TABLE command sequence with new data parameters, because when using TRUNCATE TABLE, all system references to the table will remain intact, while in the latter case, all references to this table from other tables, replication rules, table access rights etc. will be lost.

Data Manipulation Language (DML)

Delete

Deletes rows from a table.

Syntax

```
<Delete query> ::= DELETE FROM
                  [<User name>.] {<Table name> | <View name>}
                  [[AS] <Correlation name >]
                  [JOIN <Table name>[ ,... ] ]
                  [ <Where clause> ]
                  [WAIT|NOWAIT];
```

Remarks

- 1) Let T be a table specified by <Table name>.
- 2) The user must have the DELETE privilege on T.
- 3) T can not be a read-only table.

Cases:

- If <Where clause> is not specified, then all rows are deleted from T.
- If <Where clause> is specified, that condition is applied to each row of T. All rows of T for which the <Condition> is true are deleted.

Examples

```
1 DELETE FROM Person;
2 DELETE FROM Auto WHERE Color = 'GREEN';
3 DELETE FROM Auto JOIN Person WHERE auto.id=person.id;
```

Insert

Inserts a new row into a table.

Syntax

```
<Insert query> ::= INSERT INTO
                  [<User name>.] {<Table Name>|<View name>}
                  [[AS] <Correlation name >]
                  DEFAULT VALUES
                  [ (<Column Name>
                    { VALUES(<Value>[ , ... ] )
                    | <Subquery> }
                  [WAIT | NOWAIT];

<Value> ::= <Value expression>
          | NULL
          | DEFAULT
          | ?
          |:<SQL parameter's name>
          | EXTFILE (<File name>[,<Filter name>])
```

<SQL parameter's name>	::=	<Identifier>
<File name>	::=	NULL ? <Character literal>
<Filter name>	::=	<Identifier>

Remarks

- 1) The user must have the INSERT privilege on <Table>.
- 2) Let T be the table specified by the <Table>.
- 3) T must be a base table or updatable view.
- 4) Each <Column Name> must specify a column of T. No column of T can be specified more than once. ROWID is not allowed.
- 5) If no <Column Name> is specified, the implicit column list includes all columns of T sorted by their positions in T.
- 6) If one or more <Column Name>s are specified, the number of <Value>s must equal the number of <Column Name>s. The i-th <Value> will be inserted into the i-th <Column Name>.
- 7) If a <Subquery> is included, the number of columns of the table specified by this expression must equal the number of <Column Name>s included. The i-th <Column Name> corresponds to the i-th column of a table specified by <Subquery>.
- 8) If a <Value> is not null, the data types of <Column Name>s and <Value>s must match or can be converted to type of corresponding column.

Auto conversion is made for following data types:

- SMALLINT, INT, BIGINT, DECIMAL – between themselves;
 - REAL, DOUBLE – between themselves;
 - SMALLINT, INT, BIGINT, DECIMAL is converted to REAL, DOUBLE;
 - All CHARACTER types between themselves.
- 9) If the DEFAULT VALUES option is selected, default values will be inserted in all columns when a row is added. If DEFAULT value is not specified for column then NULL value is added.

Example

```
CREATE TABLE tab1 (i INT AUTOINC, c CHAR (10) DEFAULT '???' , d
DATE DEFAULT SYSDATE, n DEC);
INSERT INTO tab1 DEFAULT VALUES;
INSERT INTO tab1 DEFAULT VALUES;
SELECT * FROM tab1;
| 1 | ??? | 28.04.3003:13:10:15.00 | NULL |
| 2 | ??? | 28.04.3003:13:10:15.00 | NULL |
```

10) Let:

- C be a column of T;
- L be the length of C;
- V be a value to be inserted in C;
- M be the length of V.

11) If the data type of C is a character or byte string:

- and if $L = M$, V will be inserted in C .
 - and if $M < L$, the 1st M bytes of V are inserted in C and $L - M$ bytes of C are padded with: spaces, for character strings, or spaces, for byte strings.
- 12) If data type of C is DECIMAL, and if the value of scale is greater than specified for C , then the scale will be rounded. In such case the value of precision is greater than that specified for C and an error condition is generated.
 - 13) If the data type of C is AUTOINC (except when AUTOINC WITH RANGE), the i -th V must be greater than any pre-existing value in the i -th C .
 - 14) If value of AUTOROWID column is specified, ROWID of inserted row is assigned to specified value of column. If specified value is taken then error is generated.
 - 15) If value of AUTOROWID column is not specified, the value of AUTOROWID column of inserted row is equal ROWID of inserted row.
 - 16) If a <Value> is null and the data type of column is EXTFILE then EXTFILE(NULL) is used for inserting.

Example

```
CREATE TABLE tab1 (i INT, ext1 EXTFILE ROOT 'c:\linter\ext',
ext2 EXTFILE);
INSERT INTO tab1(i, ext1, ext2)VALUES (1,NULL,EXTFILE(NULL));
```

Update

Updates rows of a table.

Syntax

```
<Update query> ::= UPDATE
                 [<User name>.]
                 {<Target Table name> | <Target View name>}
                 [[AS] <Correlation name >]
                 [JOIN {<Table name> | <View name>}
                 [[AS] <Correlation name> [, ... ] ]
                 SET <Set list> [ WHERE <Condition> ]
                 [WAIT|NOWAIT];

<Set list>      ::= <Set clause> [, ... ]
<Set clause>   ::= <Column name> =
                 {<Value expression> | <Subquery>
                 | ?
                 | : <Parameter name>
                 | NULL
                 | EXTFILE(NULL | ?|
                 <File specification>[,<Filter name>])}
```

Remarks

- 1) Let T be a table specified by <Target Table name>.
- 2) The user must have the UPDATE privilege on <Target Table name>.
- 3) The user must have the SELECT privilege on <Table name> in JOIN clause.
- 4) T must be a base table or updatable view.

- 5) No <Column name> in a <Set clause> can be specified more than once, but you may use multiple <Set clause>s.
- 6) The scope of <Table name> following UPDATE is the entire <Update query>.
- 7) If <Condition> is omitted, all rows of T are updated;
- 8) If <Condition> is specified, all rows of T for which the result of the <Condition> are true, are updated.
- 9) <Column name> must be a column of the <Target table> or <Target View>. AUTOROWID, AUTOINC, ROWID, ROWTIME, BLOB and ROWNUM are not updateable columns.
- 10) If '?' or <Parameter name> is specified, the data type of parameter is the same as data type of the respective column.
- 11) <Subquery> must return a single value.

Examples

```

1 CREATE TABLE tab1 (type CHAR(10), id INT);
  INSERT INTO tab1 VALUES ('System', 1);
  UPDATE tab1 SET type=NULL, id=id+3 WHERE id = 1;
  SELECT * FROM tab1;
  | NULL | 4 |

2 UPDATE auto JOIN person SET auto.color='GREEN' WHERE
  auto.id=person.id;

```

Massive Data Append

The massive data append is only available in the Call interface using PUTM. It is initiated by the SQL command START APPEND and is terminated by the SQL command END APPEND. Only COMMIT ROLLBACK and CLOSE are allowed between START APPEND and END APPEND.

Start Append

Syntax

```

<Start massive append> ::= START APPEND INTO
                          [<User name>.] {<Table name> | <View Name>}
                          [BYTE | CHAR] [ (<Column Name >
                          [BYTE | CHAR| 'date format']
                          [,...])]
                          [VALUES (? [,...])]
                          [WAIT | NOWAIT];

```

Remarks

- 1) [BYTE | CHAR] following <Table> sets the conversion mode for all columns in the table. If BYTE is specified after <Table>, column data will not to be converted. The programmer must insure that data sent with PUTM is correctly formatted. By default, conversion uses CHAR format.
- 2) The specification [column name [BYTE | CHAR | 'date format']] sets the conversion mode for one specific column. By default, conversion uses CHAR format.

- 3) The column conversion mode overrides the conversion mode set for the entire table.
- 4) The data type of column cannot be a BLOB type.
- 5) VALUES clause sets the list of parameters. The number of parameters ('?' symbol) must equal the number of appended columns.

End Append

Syntax

```
<End massive append> ::= END APPEND
                        [INTO [<User name>.]<Table Name>
                        |<View Name>];
```

Remarks

- 1) <Table name>/<View Name> should reference to <Table Name>/<View name> used in START APPEND statement.

Positioned Delete and Update

Positioned Deletion

Deletes the specified row of a table contained in the current CURSOR (result of the current SELECT query).

Syntax

```
<Positioned delete> ::= DELETE FROM
                        [<User name>.] {<Table name> | <View name>}
                        [[AS] <Correlation name>]
                        WHERE CURRENT OF {CURSOR | <Cursor name>}
                        [WAIT | NOW AT];
```

Remarks

- 1) User must have the DELETE privilege on <Table name>.
- 2) The <Positioned delete> query by using 'WHERE CURRENT OF CURSOR' clause can be submitted only after submitting SELECT query through the same connection.
- 3) In case of using WHERE CURRENT OF <Cursor name>, cursor name must be set by SETO (CALL-interface command), DECLARE CURSOR (Embedded SQL) or any respective command of any other interfaces (ODBC and so on).
- 4) Let T be the table specified by <Table name>. T must be the table specified in the first <From clause> of the current SELECT query.
- 5) T and the result of the current SELECT must be updatable.
- 6) Positioned delete statement deletes the current row of the specified cursor. The current row is the last row fetched from the cursor (result set).

Positioned Update

Updates a row of a table contained in the current CURSOR (result of the current SELECT query).

Syntax

```

<Position_Update> ::= UPDATE
                    [ <User name>.]
                    { <Table name> | <View name> }
                    [[AS] <Correlation name>]
                    SET<Position set clause>[ ,... ]
                    WHERE CURRENT OF
                    {CURSOR | <Cursor name> }
                    [WAIT | NOW AT];

<Position set clause> ::= <Column name>= {<Value expression>
                    | <Subquery>
                    | NULL
                    | EXTFILE(NULL | ? | <File specification> [, <Filter
                    name>])}
  
```

Remarks

- 1) User must have the UPDATE privilege on <Table name>.
- 2) The <Positioned update> query by using 'WHERE CURRENT OF CURSOR' clause can be submitted only after submitting SELECT query through the same connection.
- 3) <Positioned update> statement updates the current row of the specified cursor. In case of using WHERE CURRENT OF <Cursor name>, cursor name must be set by SETO (CALL-interface command), DECLARE CURSOR (Embedded SQL) or any respective command of any other interfaces (ODBC and so on). The current row is the last row fetched from the cursor (result set).
- 4) Let T be the table specified by <Table name>. T must be the table specified in the first <From clause> of the current SELECT query.
- 5) T and the result of the current SELECT must be updatable.
- 6) The <Value expression> in a <Positioned set clause> can not include an aggregate functions.
- 7) Each <Column name> in <Positioned update> must be unique.
- 8) The scope of <Table name> is the entire <Positioned update>.
- 9) <Column name>s must be the names of the columns of T.
- 10) If <Value expression> contains a reference to a column of T, that reference represents the value of the specified column in the row before updating.
- 11) A row is updated by
 - Creating a copy of the initial row as a candidate row.
 - For each <Positioned set clause>, the value of the specified column in the candidate row is changed to the newly specified value.
 - The initial row is replaced by the candidate row.

- 12) If T is a view defined by a WITH CHECK OPTION clause and the query defining this view contains a WHERE clause and the <Condition> of the WHERE clause is false for the candidate row, an exception is generated.

Lock and Unlock Table

Lock Table

Locks a table.

Syntax

```
<Table locking>          ::= LOCK TABLE
                           [ <User name>.{ <Table name>}
                           [IN <Access mode> MODE]
                           [ WAIT | NOWAIT];
<Access mode>           SHARE | EXCLUSIVE
```

Remarks

- 1) <Table name> must refer to a base table, not a view.
- 2) If <Access mode> is omitted, EXCLUSIVE is the default.
- 3) If [WAIT | NOWAIT] is omitted, WAIT is the default.
- 4) .
- 5) SHARE <Access mode> allows other DB users read-only access to a locked table. EXCLUSIVE <Access mode> denies any access to the locked table to all other DB users.
- 6) The [WAIT | NOWAIT] options define what happens when a table cannot be locked. WAIT means to wait as long as necessary until the table can be locked. NOWAIT means to abandon the locking attempt.
- 7) Table lock is turned off by the UNLOCK, COMMIT, and ROLLBACK operators.

Unlock Table

Unlocks a table.

Syntax

```
<Table unlocking>       ::= UNLOCK TABLE
                           [ <User name>.{ <Table name>};
```

Remarks

- 1) Only the user who has locked the table can unlock the table.

Transaction Management

Set Savepoint

Creates intermediate savepoint in the current transaction.

Syntax

```
<Create Savepoint> ::= SET SAVEPOINT <Savepoint name>;
```

Remarks

- 1) <Savepoint name> must be unique within given transaction.

Commit

Saves database changes performed by the current transaction.

Syntax

```
<Commit> ::= COMMIT [WORK]  
[ TO SAVEPOINT[<Savepoint name>]];
```

Remarks

- 1) If <Savepoint name> is specified, it must refer to an existing savepoint created by the SAVEPOINT operator.
- 2) If there are no intermediate savepoints, COMMIT saves all DB changes made since the beginning of the transaction.
- 3) If there are one or more intermediate savepoints, COMMIT saves those DB changes that have been made for the current transaction since the last COMMIT or ROLLBACK to a savepoint.
- 4) If TO <Savepoint name> is omitted, the current transaction implicitly terminates and a new transaction begins.
- 5) On completion of COMMIT:
 - ROLLBACK is no longer possible for the committed transaction.
 - Locks that were set during the transaction are cancelled.
 - Intermediate savepoints created in the transaction are removed.
- 6) The WORK is supported for compliance with standard SQL.
- 7) COMMIT and COMMIT WORK are equivalent.

Rollback

Cancels, rolls back, database changes performed by the current transaction.

Syntax

```
<Rollback> ::= ROLLBACK[WORK]  
[ TO SAVEPOINT[<Savepoint name>]];
```

Remarks

- 1) If <Savepoint name> is specified, it must refer to an existing savepoint created by the SAVEPOINT operator.
- 2) If there are no intermediate savepoints, ROLLBACK cancels all DB changes made since the beginning of the transaction.
- 3) If there are one or more intermediate savepoints, ROLLBACK cancels those DB changes that have been made for the current transaction since the last COMMIT or ROLLBACK to a savepoint.
- 4) If TO <Savepoint name> is omitted, the current transaction implicitly terminates and a new transaction begins.
- 5) On completion of ROLLBACK:
 - The current transaction's previous changes are removed.
 - Locks that were set during the transaction are cancelled.
 - Intermediate savepoints created in the transaction are removed.

Read only

Establishes the current transaction as a read-only.

Syntax

```
<Read-only mode> ::= SET TRANSACTION READ ONLY;
```

Remarks

- 1) All subsequent queries in that transaction can see only changes committed before the beginning of the transaction.
- 2) Read-only mode is canceled by COMMIT/ROLLBACK or close current session.

Triggers**Create Trigger**

Creates a trigger.

Syntax

```
<Create Trigger> ::= CREATE [OR REPLACE] TRIGGER
                    [<User name>.]<Trigger name>
                    <Action time> <Operations>
                    ON [<User name>.]<Table name>
                    [FOR EACH {ROW | STATEMENT} ]
                    [OLD [AS] <Old row correlation name >]
                    [NEW [AS] <New row correlation name >]
                    EXECUTE <Code>;

<Action time> ::= BEFORE | AFTER
<Operations> ::= <Operation> | <Operations> OR <Operation>
<Operation> ::= INSERT
               | DELETE
```

```

<Code>                ::=  | UPDATE [[OF <Column> [ ,... ] ]
                        <Procedure body>

```

Remarks

- 1) Several triggers can be created for every action on any table.
- 2) Only owner of the table can create trigger on this table.
- 3) <Operation> specifies event which fire a trigger:
 - DELETE – invoked whenever a row of associated table is deleted.
 - INSERT – invoked whenever a new row is inserted into the table associated with the trigger.
 - UPDATE – invoked whenever a row of the associated table is updated. If OF keyword is omitted trigger is fired whenever an UPDATE statement changes a value in any column of the table. If OF keyword is specified trigger is fired whenever an UPDATE statement changes a value in one of the columns specified after OF.
- 5) Specify BEFORE to fire the trigger before executing the <Operation>. For row triggers, the trigger is fired before each affected row is changed.
- 6) Specify AFTER to fire the trigger after executing the <Operation>. For row triggers, the trigger is fired before each affected row is changed.
- 7) FOR EACH clause specifies ROW or STATEMENT level of trigger:
 - ROW – fires a row trigger once for each row that is affected by the <Operation>.
 - STATEMENT – fires a trigger for each statement of <Operation>.
- 9) Triggers of the different <Action time> are fired by following order:
 - BEFORE ... FOR EACH STATEMENT...;
 - BEFORE ... FOR EACH ROW...;
 - AFTER ...FOR EACH ROW ...;
 - AFTER ... FOR EACH STATEMENT ...;
- 10) 'OLD AS' and 'NEW AS' clauses specify correlation names to refer specifically to old and new values of the current row.
- 11) The default correlation names are OLD and NEW.
- 12) OLD AS clause allows you to refer to the values in a row prior to an update or delete.
- 13) NEW AS clause allows you to refer to the inserted or updated values.
- 14) <Procedure body> should be written on Storage procedure language.

Drop Trigger

Deletes a trigger.

Syntax

```

<Delete Trigger>      ::=  DROP TRIGGER
                        [<User name>.<Trigger name>;

```

Remarks

- 1) Only the owner of the trigger can delete the trigger.

Alter Trigger

Changes trigger status.

Syntax

```
<Change Trigger Status> ::= ALTER TRIGGER
                             [<User name>.<Trigger_Name>
                             {ENABLE | DISABLE};
```

Remarks

- 1) ENABLE switches trigger to active state.
- 2) DISABLE switches trigger to inactive state.
- 3) Changing trigger status doesn't change its code.

Stored Procedures**Create Procedure**

Creates a procedure.

Syntax

```
<Create Procedure> ::= CREATE [OR REPLACE]
                    <Procedure code>;
```

Remarks

- 1) See document "Stored Procedure Functions".

Example

```
create procedure aaa( )
result cursor(id int, make char(20)) for debug
Declare
var c typeof (result);
exception notab for 2202;
Code
open c for direct "select id,make from auto;";
return c"
Exceptions
    when 2202 then
        print("no such table!\n");
// debug print end;
```

Alter Procedure

Modifies the stored procedure code.

Syntax

```
<Modify Procedure> ::= ALTER <Procedure code>;
```

Drop Procedure

Deletes the stored procedure.

Syntax

```
<Delete Procedure> ::= DROP PROCEDURE  
[<User name>.] <Procedure name>;
```

Remarks

- 1) Only the owner of the stored procedure can delete the stored procedure.

Execute

Executes the stored procedure.

Syntax

```
<Execute Procedure> ::= EXECUTE  
[<User name>.] <Procedure name>;  
(<Parameter list>) [AS OWNER];  
  
<Parameter list> ::= (<expression>[ ,... ]  
| <variable name> [ ,... ] )
```

Remarks

- 1) Omitted parameters are replaced by commas.
- 2) When actual parameters are omitted, the called procedure receives default value parameters. This is the same as when a stored procedure is called from within another stored procedure.
- 3) Logical value representation (strings are case sensitive):
 - TRUE = 1 or "true",
 - FALSE= 0 or "false".

Data Replication

Create Replication Server

Defines data replication server.

Syntax

```
<Replication server> ::= CREATE [OR REPLACE]
                        {NODE|SERVER} <DB Server name>
                        [LOCAL | REMOTE];
```

Remarks

- 1) <DB Server name> is identifier <= 8 characters in length.
- 2) The system table REPLICATION must exist.
- 3) <DB Server name> must identical to one of Linter-server names in 'nodetab' file for given Relex DB Linter series server.
- 4) You must have DB privileges for creating replication server.

Drop Replication Server

Deletes replication server from system table.

Syntax

```
<Delete Replication server> ::= DROP {NODE|SERVER}
                                <DB Server name> [CASCADE];
```

Remarks

- 1) <DB Server name> is identifier <= 8 characters in length.
- 2) <DB Server name> must exist in SERVERS system table.
- 3) You must have DB privileges for deleting replication server.
- 4) If CASCADE keyword is specified all replication rules referenced to dropped replication server are deleted.
- 5) If CASCADE keyword is omitted and such referential replication rules exist, then Linter returns an error and doesn't drop a replication server.

Create Replication Rule

Defines a data replication rule.

Syntax

```
<Replication Rule> ::= CREATE [OR REPLACE]
                       REPLICATION RULE <Rule name>
                       FOR <Local table name>
                       [TO <Remote table name>]
                       ON {NODE| SERVER} <Node name>
                       [USER <'User name'>]
```

```

[PASSWORD <'Password'> ]
[ENABLE | DISABLE]
[SYNC | ASYNC]
[<Priority rule list>]
[<Calculate rule list>]
[COLUMN (<Column name>[,...])
[<Priority rule list>]
[<Calculate rule list>] [ ...];
<Priority rule list> ::= PRIORITY
                        {SECOND
                         | FIRST
                         | NEW
                         | OLD
                         | WEIGHT
                         | DEFAULT}
<Calculate rule list> ::= CALCULATE
                        { NONE
                         | MAX
                         | MIN
                         | AVG
                         | DIFFERENCE
                         | DEFAULT}

```

Remarks

- 1) <Rule name> is a string <= 66 characters in length.
- 2) <Local table name> and <Remote table name> must be names of base tables.
- 3) If ENABLE is specified the replication rule is available for use.
- 4) If DISABLE is specified the replication rule is not available for use.
- 5) The rule is ENABLED by default.
- 6) SYNC specifies a synchronous mode.
- 7) ASYNC is default and specifies asynchronous mode.
- 8) PRIORITY and CALCULATE clauses specify a conflict resolution rules. You can specify PRIORITY and CALCULATE for table and for single column.
- 9) Specify COLUMN clause to set conflict resolution rules for individual columns.
- 10) System tables SERVERS and \$\$\$REPL must exist.
- 11) <User name> and <Password> specify the remote node user name and the password.
- 12) The local and remote table structures must be identical.
- 13) You cannot replicate tables containing EXTFILE data.
- 14) Both the local and remote table must have a PRIMARY KEY.
- 15) To create replication rule on table you must be owner of that table or have SELECT, DELETE, INSERT, UPDATE privileges.

Alter Replication Rule

Changes replication rule's status or password.

Syntax

```

<Alter Rule> ::= ALTER
                REPLICATION RULE <Rule name>
                [PASSWORD <'Password'> ]
                [ENABLE | DISABLE]
                [<Priority rule list>]
                [<Calculate rule list>]
                [COLUMN (<Column name>[,...])]
                [<Priority rule list>]
                [<Calculate rule list>] [ ...];

<Priority rule list> ::= PRIORITY
                        {SECOND
                         | FIRST
                         | NEW
                         | OLD
                         | WEIGHT
                         | DEFAULT}

<Calculate rule list> ::= CALCULATE
                        {NONE
                         | MAX
                         | MIN
                         | AVG
                         | DIFFERENCE
                         | DEFAULT}

```

Remarks

- 1) <Rule name> is a string <= 66 characters in length.
- 2) <Rule name> must exist in \$\$\$REPL system table.
- 3) <Password> specify the remote node password.
- 4) ENABLE and ASYNC are specified by default.
- 5) To alter replication rule on table you must be owner of that table or have SELECT, DELETE, INSERT, UPDATE privileges.
- 6) PRIORITY and CALCULATE clauses specify a conflict resolution rules. You can specify PRIORITY and CALCULATE for table and for single column.
- 7) Specify COLUMN clause to set conflict resolution rules for individual columns.

Drop Replication Rule

Deletes replication rule.

Syntax

```

<Drop Rule> ::= DROP REPLICATION RULE <Rule name>;

```

Remarks

- 1) <Rule name> is a string <= 66 characters in length.
- 2) <Rule name> must exist in \$\$\$REPL system table.

- 3) To drop replication rule on table you must be owner of that table or have SELECT, DELETE, INSERT, UPDATE privileges.

Synchronization Replication Rule

Synchronizes replication rules for local and remote tables. Synchronization of replication rule usually is used when the new replication rule is created.

Syntax

```
<Rule synchronization> ::= SYNCHRONIZE
                             REPLICATION RULE <Rule name>;
```

Remarks

- 1) <Rule name> is a string <= 66 characters in length.
- 2) <Rule name> must exist in \$\$\$REPL system table.
- 3) To SYNCHRONIZE replication rule on table you must be owner of that table or have SELECT, DELETE, INSERT, UPDATE privileges.

Events

You can use event mechanism through SQL language constructions.

Create Event

Declares a new event.

Syntax

```
<Create Event> ::= CREATE EVENT <Event name>
                 [AUTORESET] AS <Event type>;
<Event type>   ::= {<Data check>|<DB object update>}
<Data check>   ::= <Query expression>
<DB object update> ::= <Operation> ON <Table name>
<Operation>    ::= DELETE | UPDATE | INSERT
```

Remarks

- 1) All events must have unique names.
- 2) <Table name> must refer to a base table.
- 3) Created event exists until it is deleted by DROP EVENT or Linter kernel shutdown.
- 4) <Check query> defines a query that must be executed automatically when any table accessed in the query is actually updated.
- 5) To create an event you must have at least RESOURCE level access.
- 6) If <Event type> is applied to the remote object, it will be seen only in the program that created the event. On the remote node this event will be unknown.
- 7) For event with AUTORESET clause the event status will be reset after event status inquirer.

Drop Event

Deletes the event.

Syntax

```
<Delete Event> ::= DROP EVENT <Event name>;
```

Remarks

- 1) An event can be deleted by its owner or by the user who has DBA level access.
- 2) An event is automatically deleted when the Linter kernel is shutting down.
- 3) An event can't be deleted if a WAIT EVENT is pending.

Set Event

Inserts an event into the event queue.

Syntax

```
<Queue Event> ::= SET EVENT <Event name>;
```

Remarks

- 1) There is a single, common queue for all events.
- 2) An event is considered as happened independently of whether the transaction was ended by COMMIT or ROLLBACK and is inserted into the queue in the following cases:
 - If an event was created with the use of <Check query>, it is queued when the specified query returns a non-empty answer. <Check query> is executed every time the tables accessed by the source query are updated.
 - If an event was created with the operation type specified, it is queued after execution of the specified operation on the specified table when a non-empty count of processed rows is returned.
 - In both a and b , above, if the event is already in the queue, it isn't inserted into the queue again. If it was in the queue and <Check query> or a table update query return an empty answer, the event is removed from the queue.
- 3) An event can be explicitly inserted into the queue either by its creator or by a user who has DBA level access.

Wait Event

Waits for the event.

Syntax

```
<Wait for Event> ::= WAIT EVENT <Event expression>;
```

Remarks

- 1) <Event expression> should contain <Event name>s , logical operation (OR, AND, NOT), and parenthesis.

Example

```
CREATE EVENT evn1 AS DELETE, INSERT ON tab1;
```

```
CREATE EVENT evn2 AS UPDATE ON tab2;  
CREATE EVENT evn3 AS SELECT * FROM tab3;  
WAIT EVENT (env1 AND env3) OR (env2 AND env3);
```

- 2) When an event occurs, control is transferred to the program waiting for the event. The event isn't removed from the queue.

Get Event

Returns event status.

Syntax

```
<Get Event Status> ::= GET EVENT <Event name>;
```

Remarks

- 1) If the NOT modifier is omitted, code zero is returned to the program if the event is in the queue, code 2 is returned if it is not in the queue.
- 2) The event isn't removed from the queue in either case.

Clear Event

Removes an event from the queue.

Syntax

```
<Clear Event from Queue> ::= CLEAR EVENT <Event name>;
```

Remarks

- 1) An event can be removed from the queue only by the creator of the event or by a user who has DBA level access.

Users And Roles

In large databases, where there maybe thousands of users and tables, it is difficult to specify privileges for every user on every table. The use of roles simplifies that procedure.

A role is a named set of rights on a set of data base objects. Owners of some database objects assign access privileges on such objects to the role (in the same way they assign these privileges to users). The role's owner assigns this role to a user or a group of users. The user who has been assigned a role then has all of the table privileges associated with his role.

The following capabilities for working with users and roles are presented in SQL:

- Create and delete user and change password and group;
- Create and delete role;
- Grant and revoke table privileges.

Create User

Creates a new DB user.

Syntax

```
<Create User>          ::=  CREATE [OR REPLACE]
                        USER <User name>
                        [IDENTIFIED BY <password>]
                        [GROUP <Group name>];
<User name>           ::=  Literal not to exceed 66 characters
<Group name>          ::=  Literal not to exceed 66 characters
<password>            ::=  Literal not to exceed 66 characters
```

Remarks

- 1) User creating a user must have DBA level access.
- 2) <User name> must be unique among all DB users.
- 3) By default, password is 18 spaces.
- 4) By default, new user has CONNECT level access.

Drop User

Deletes a user.

Syntax

```
<Delete User>          ::=  DROP USER <User name> [CASCADE];
```

Remarks

- 1) To delete a user you must have DBA level access.
- 2) CASCADE is used for deleting all objects of the <User name> and all dependencies.
- 3) When a user is deleted, all his privileges and role assignments are deleted too.
- 4) The user which owns some sequences should use CASCADE option.

Alter User


Change user password or group.

Syntax

```
<Alter User> ::= ALTER USER [<User_name>]
                [IDENTIFIED BY <password> ]
                [GROUP <Group name>]
                [PASSWORD LENGTH
                {MIN<Value> | UNLIMITED}];
```

Remarks

- 1) A user password can be changed only by a DBA or by the user himself.
- 2) If <User name> is omitted, the user who issued the command is assumed.
- 3) Changing <User name> and <Group name> can be specified in the same statement.

 A DBA can abuse his authority by changing a user's password to gain access to information in that user's tables. The only protection for this problem is that the user whose password was changed will be unable to logon to the system and, presumably, will notify someone. One partial roadblock to this type of abuse is that the evil DBA cannot restore the user's old password.

Create Role

Creates a new role.

Syntax

```
<Create Role> ::= CREATE [OR REPLACE] ROLE <Role name>;
```

Remarks

- 1) <Role name> must be unique among all role and usernames.
- 2) The user creating a role must have at least RESOURCE level access.

Drop Role

Deletes the existing role.

Syntax

```
<Delete Role> ::= DROP ROLE <Role name>;
```

Remarks

- 1) Only the creator of the role can delete it.
- 2) When the role is deleted, all privileges assigned to this role and all its assignments to users and other roles are also deleted.

Grant Role

Assigns a role to one or more users.

Syntax

```
<Role assignment> ::= GRANT ROLE <Role name>
                        TO {<User name>[ ,... ]|PUBLIC};
```

Remarks

- 1) Only the user who created the role can assign it.

Revoke Role

Cancels role assignment.

Syntax

```
<Revoke Role Assignment> ::= REVOKE ROLE <Role name>
                              FROM
                              {{<User name>
                              | <Role name>}[ ,... ]
                              | PUBLIC};
```

Remarks

- 1) Only the user who created the role can cancel its assignment.

Synonyms

Table name synonyms can simplify writing SQL queries by shortening the object's name. There are two types of synonyms, public and individual. When creating or deleting synonyms, distinguishing between them is required. Using the PUBLIC keyword is the method used to make the distinction.

Linter allows multiple synonyms for table names. Different users may have different synonyms for the same table.

Create Synonym

A synonym can contain up to 66 symbols. If it contains non-alphanumeric symbols, it must be enclosed in double quotes.

Syntax

```
<Create synonym> ::= CREATE [ OR REPLACE]
[PUBLIC] SYNONYM <Synonym>
FOR [ <User name>. ] <Table name>;
```

Remarks

- 1) The specified table must exist at the time the synonym is created.
- 2) A synonym must be unique among the synonyms, table names, and view names created by the same user. If PUBLIC is specified, the synonym cannot have the same name as any other public synonym.
- 3) If PUBLIC is specified, the created synonym is available to all users who do not themselves have a table or synonym with the same name. If PUBLIC is not specified, any other users can use this synonym with the construct: <Creator's name>.<Synonym>.
- 4) To create a PUBLIC SYNONYM you must have DBA level access.

Drop Synonym

Syntax

```
<Delete synonym> ::= DROP [ PUBLIC ] SYNONYM <Synonym>;
```

Remarks

- 1) If the PUBLIC keyword is specified, then a public synonym is to be deleted. Otherwise, a synonym created by the user is to be deleted.
- 2) Any user can delete a public synonym. Only the user who created a non-public synonym can delete it.
- 3) The deletion of a synonym does not affect the table to which the deleted synonym refers.
- 4) To delete a PUBLIC SYNONYM you must have DBA level access.

Example

```
DROP PUBLIC SYNONYM PUBLIC_AUTO;
```

User Queries Trace File Management

Syntax

```
<Trace> ::= SET LOG { OFF | BRIEF | DEFAULT | FULL|ON };
```

Remarks

- 1) OFF stops.
- 2) When the role is deleted, all privileges assigned to this role and all its assignments to users and other roles are also deleted.
- 3) User requests are written in the linter.log file.
- 4) The OFF option disables the tracing of the user request to Linter.
- 5) The BRIEF option enables the protocol of the
- 6) user request tracing in the same way as the launching of Linter using /LOGQUERY key.
- 7) The DEFAULT option enables the protocol of user request tracing in the same way as the launching of Linter using /LOG key without /LOGQUERY and /LOGALL commands.
- 8) The FULL option enables complete protocol of the user request tracing in the same way as launching Linter using /LOGALL command.

Transaction Control

Syntax

<Transaction control> ::= SET TRUE COMMIT { ON | OFF };

Remarks

- 1) The ON option is used by default.
- 2) During the transaction process all changes performed are committed in the user request tracing.
- 3) If the transaction is completed with COMMIT command, then all changes made by the user will exist in the database.
- 4) If the OFF option is specified, then the system does not guarantee all changes made by user to exist in the database, but it does provide more efficient performance.

Job Sharing Execution

Request of time consumption for the query execution.

Syntax

<Time consumption>	::=	<Using index> <No index>
<Using index>	::=	SET INDEX QUANT <Value>
<No index>	::=	SET ROW QUANT <Value>
<Value>	::=	Unsigned INT

Remarks

- 1) <Value> is the number of time consumption used.
- 2) User is required to have DBA privileges.

Set User Priority

Function sets and changes the user priority.

Syntax

```
<Set user priority>      ::=  SET PRIORITY FOR <User Name>
                           <Priority level> [, ...];
<Priority level>         ::=  { BASE=<value>
                              | MAX=<value>
                              | RANGE=<value> }
<User name>             ::=  <Identifier>
<Value>                 ::=  Unsigned INT
```

Remarks

- 1) <Value> – unsigned integer between 0 and 255.
- 2) MAX=<value> parameter sets the maximum allowed user priority (highest range priority).
- 3) BASE=<value> parameter sets default user priority. RANGE=<value> parameter sets the minimum allowed user priority (lowest range priority). If MAX, BASE, and RANGE parameters are set, then the system assumes MAX = BASE (does not exceed), RANGE = BASE (decreased to 0). If the user sets all three parameters, and the new request sets only BASE, then automatically RANGE and MAX are modified.
- 4) Priority values in 0-99 range allow dynamic changes of priority by the user. During each cycle of time consumption each channel gets its own share of time consumption. The channel location in the queue is set by its activation time and not by its priority.
- 5) Priority values in the 100-199 range sets the time consumption of user query during the cycle.
- 6) Priority values in the range 200-249 provide automatic time consumption of user queries and prioritize system process and system queries.
- 7) Priority value in the 250-256 range is a reserved group.

Sequences

Sequence Creation

Creation of public and private sequence.

Syntax

```

<Sequence creation> ::= CREATE [OR REPLACE]
                        SEQUENCE [PUBLIC]
                        [<User name>.<Sequence name>
                        [ START WITH <Initial value> ]
                        [ INCREMENT BY <Interval> ]
                        [MINVALUE <Beginning range> ]
                        [MAXVALUE <Ending range> ];
<Initial value>      ::= INTEGER | RANGE=<value>
<Interval>           ::= INTEGER
<Beginning range>   ::= INTEGER
<Ending range>      ::= INTEGER
<User name>         ::= <Identificator>
  
```

Remarks

- 1) If PUBLIC option is set, then the sequence is accessible to all database users.
- 2) If any user has a sequence with <sequence name>, then the PUBLIC SEQUENCE with the same <sequence name> is not accessible for that user.
- 3) <Sequence name> must be unique for each given user.
- 4) <Initial value>, <interval>, <beginning range> and <ending range> – whole values in the range from -9 223 372 036 854 775 808 to +9 223 372 036 854 775 807 (data type Bigint).
- 5) <Initial value>, <interval>, <beginning range>, and <ending range> values are required to logically match one another.
- 6) The user is required to have DBA privileges to create PUBLIC SEQUENCE.
- 7) Priority values in the range 200-249 provide automatic time consumption of user queries and prioritize system process and system queries.
- 8) Priority value in the 250-256 range is a reserved group.

Sequence Deletion

Deletion of public and private sequence.

Syntax

```

<Sequence deletion> ::= DROP [ PUBLIC ] SEQUENCE
                        [<User name>.<Sequence name>];
  
```

Remarks

- 1) The user must have DBA privileges to delete PUBLIC SEQUENCE.

Sequence Use

Assignment of the next and current sequence values.

Syntax

```
<Sequence value> ::= [<User name>.<Sequence name>
                        .{NEXTVAL | CURRVAL};
```

Remarks

- 1) NEXTVAL – get the next sequence value.
- 2) CURRVAL – get the current sequence value, which is the value returned by the last reference to NEXTVAL .
- 3) <Sequence value> function can be used:
 - SELECT queries – pseudocolumn in the WHERE table expression;
 - UPDATE queries – in the SET clause and in the WHERE table expression;
 - INSERT queries – in the VALUES clause;
 - INSERT queries – VALUES table expression;
 - DELETE queries – WHERE table expression.
- 4) Before you use CURRVAL for a sequence in your session, you must first initialize the sequence with NEXTVAL
- 5) Within a single SQL statement, sequence will be incremented only once for each row. If there are more than one NEXTVAL for a sequence, sequence returns the same value for all occurrences of NEXTVAL.

Code Page Use

Code page Client application can work in different code tables only for the 6.0. version. User can set user code pages. Linter stores the character data using the code pages set by the user. These code pages are seed only for a single byte pages except MBCS and UTF8 which will be converted to UNICODE. The data of the single byte pages is received without translation.

Translation is used for the direct conversion between two single byte code pages (set table conversion). In the case where translation is not needed, code pages are generated automatically in UNICODE. In this case any missing characters are replaced by'?'.

Code Page Creation

Creation of new code pages in the database.

Syntax

```

<Code page> ::= CREATE CHARACTER
                SET <Code page name>
                [ WINDOWS_CODE
                <Windows Code page code> ]
                [ NONEUC | EUC | UTF8 ]
                [[ PLAN <Plan number >]
                PAGE <Page number>]
                EXTERNAL
                (<Code page content>);

<Code page name> ::= <Identificator>
<Windows Code page code> ::= Unsigned INT
<Plan number> ::= Unsigned INT
<Page number> ::= Unsigned INT
<Code page content > ::= 1536 byte

```

Remarks

- 1) <plan number> and <page number> options are used only when creating MBCS code pages.
- 2) <Code page content> for the single byte code pages includes the following components: CTYPE (512 byte), WEIGHTS (256 byte), CASE (256 byte), TO_UNICODE (512 byte).
- 3) CTYPE – used only for single byte code pages and is the character array (i.e. Value, character, symbol in the upper and lower case).
- 4) WEIGHTS – used only for single byte code pages and is character weight array.
- 5) CASE – used only for single by code pages and is the conversion of symbols from/to upper and lower case.
- 6) TO_UNICODE – used only for single byte code pages and the character conversion table into UNICODE (information for reverse conversion from UNICODE into code pages is automatically created during the load stage).
- 7) NONEUC – code pages are not compatible with EUC (i.e. GB (GB 2312-80. GB 12345-90 GB 7589-87. GB 7590-87. GB 8565-89.), JIS0212, KSX1001, CNS11643).

<Code page name> ::= <Identificator>

Remarks

- 1) <Code page name> sets code page which are used in the database by default.

Example

```
set database names CP866;
```

Create Translation

Create translation from one code page to another.

Syntax

```
<Translation creation> ::= CREATE TRANSLATION
                           <Translation name>
                           FOR <Source code page name>
                           TO <Target code page name>
                           EXTERNAL(<Translation array>);

<Code page name> ::= <Identificator>
<Source code page name> ::= <Identificator>
<Target code page name> ::= <Identificator>
```

Remarks

- 1) <Code page name > – translation name must be unique in the database.
- 2) <Source code page name> – code page name that is going to be translated. This code page needs to be installed in the database;
- 3) <Target code page name> – code page name into which the translation will be made. This code page needs to be installed in the database;
- 4) <Translation array> – symbol correspondence array between source and target code pages (256 byte).
- 5) Created translation name is accessible to all database users.

Translaton creaton example:

```
CREATE TRANSLATION fromCP866toCP1251 FOR CP866 TO CP1251 EXTERNAL(
hex('000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F202122
232425262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F40414243444546474
8494A4B4C4D4E4F505152535455565758595A5B5C5D5E5F606162636465666768696A6B6C6D
6E6F707172737475767778797A7B7C7D7E7F808182838485868788898A8B8C8D8E8F90919293949596979899A9B9C9D9E9FA0B6B5A3BCA5B3A7B4BBB1A
BACF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFFF8B8AABAAFBA2A1B095B7B2B9A498BE' ) ) ;
```

Deleting Code Page

Syntax

```
<Code page deletion> ::= DROP CHARACTER
                        SET <Code page name>;

<Code page name> ::= <Identificator>
```

Remarks

- 1) <Code page name> must refer to an earlier installed database code page.
- 2) User must have DBA privileges in order to delete code pages.

Example

```
DROP CHARACTER SET CP866;
```

Deleting Code Page Translation

Syntax

```
<Translation deletion> ::= DROP TRANSLATION <Translation name>;  
<Translation name> ::= <Identifier>
```

Example

```
DROP TRANSLATION fromCPi25itoCP866;
```

Linter's Extended Data Protection Features

This section briefly describes Linter's SQL data protection functions.

Audit Statements

Start Audit

Starts an audit.

Syntax

```
<Start Audit> ::= AUDIT START;
```

Stop Audit

Stops an audit.

Syntax

```
<Stop Audit> ::= AUDIT STOP;
```

Set Audit Parameters

Defines audit sizes.

Syntax

```
<Set_Audit_Parameters> ::= AUDIT {SET | CANCEL}
    <Audit_Size>;
<Audit_Size> ::= RECORD LIMIT <Row_Count>
    | DAY LIMIT <Day_Count>
<Row_Count> ::= <Significant_Expression>
<Day_Count> ::= <Significant_Expression>
```

Audit Control

Defines audit events and objects.

Syntax

```
<Audit_Control> ::= AUDIT {ENABLE|DISABLE|CLEAR}
    [<Event_name>] ON <DB_object>
    [FOR <User_Name>
    [ Y {SESSION
    | STATEMENT
    | ACCESS}} ]
    [WHEN [NOT] SUCCESS];
<Event_name> ::= <Identifier>
<DB_object> ::= <Identifier>
<User_Name> ::= <Identifier>
```

Security Groups

Create Group

Creates a new group. Linter's groups have been implemented to handle security for groups of users. Hereafter, they will be referenced simply as groups.

Syntax

```
<Create_a_Group> ::= CREATE GROUP<Group_name>
                  <Group_name>[=<Group_identifier>];
<Group_name>     ::= <Identifier>
<Group_identifier> ::= Unsigned INT
```

Rename Group

Change the name of an existing group.

Syntax

```
<Rename_a_Group> ::= ALTER GROUP<Group_name>
                  SET [<New_group_name>];
<Group_name>     ::= <Identifier>
<New_group_name> ::= <Identifier>
```

Group Access Grant

Enables data access by more than one group.

Syntax

```
<Grant_group_access> ::= GRANT ACCESS ON
                       <Grantor_group_name>
                       TO {[<Grantee_group_name>] | ALL};
<Grantor_group_name> ::= <Identifier>
<Grantee_group_name> ::= <Identifier>
```

Revoke Group Access

Revokes one group's access to another group's data.

Syntax

```
<Revoke_group_access> ::= REVOKE ACCESS ON
                        <Grantor_group_name>
                        FROM {[<Grantee_group_name>] | ALL};
<Grantor_group_name> ::= <Identifier>
<Grantee_group_name> ::= <Identifier>
```

Access Level

Data security levels allow restricting data access to users having the required security level.

Create Level

Create a new level.

Syntax

```
<Create_a_level> ::= CREATE LEVEL
                   <Level_Name> = <Level_Number>;
<Level_Name>    ::= <Identifier>
<Level_Number> ::= Unsigned INT
```

Rename Level

Change the name of an existing level.

Syntax

```
<Rename_a_level> ::= ALTER LEVEL <Level_name>
                   SET<New_level_name>;
<Level_Name>    ::= <Identifier>
<Level_Number> ::= <Identifier>
```

Managing Workstation

Create Station

Create a new workstation.

Syntax

```
<Create a Station> ::= CREATE STATION <Station_name>
                   PROTOCOL <Network_Protocol>
                   ADDRESS <Station_address>
                   [LEVEL (<RAL>,<WAL>)]
                   [<Work_time>];
<Station_name>    ::= <Identifier>
<Network_Protocol> ::= <Character_literal>
<Station_Address> ::= <Character_literal>
<RAL>             ::= <Identifier>
<WAL>            ::= <Identifier>
<Work_time>      ::= {ENABLE | DISABLE}
                   LOGIN {ALWAYS
                   | <Time_schedule>
                   |<Day_schedule>}
<Time_schedule> ::= FROM <Work_beginning_time>
                   TO <Work_end_time>
                   [FOR <Work_days>]
```

<Day_schedule>	::=	{SINCE <Beginning_date> {UNTIL <End_date>}
<Work_beginning_time>	::=	'HH:MM'
<Work_end_time>	::=	'HH:MM'
<Work_days>	::=	<Week_day> [, ...]
<Week_day>	::=	'MON' 'TUE' 'WED' 'THU' 'FRI' 'SAT' 'SUN'
<Beginning_day>	::=	'DD.MM.YYYY'
<End_day>	::=	'DD.MM.YYYY'

Drop Station

Deletes the existing workstation.

Syntax

<Station_deletion>	::=	DROP STATION <Station_name>;
<Station_name>	::=	<Identifier>

Grant Access on Unlisted Station

Enables working on a station not listed in the list of stations.

Syntax

<Grant_Access_to_ Unlisted_Station>	::=	GRANT ACCESS ON UNLISTED STATION <Station_name> TO {<Group_name> ALL};
<Station_name>	::=	<Identifier>
<Group_name>	::=	<Identifier>

Revoke Access on Unlisted Station

Disables working on a station not listed in the list of stations.

Syntax

<Revoke_Access_to_Unlisted_Station>	::=	REVOKE ACCESS ON UNLISTED STATION <Station_name> FROM {<Group_name> ALL};
<Station_name>	::=	<Identifier>
<Group_name>	::=	<Identifier>

Grant Access to Listed Station

Grants access to a listed station to the specified group or to all users.

Syntax

```

<Grant_Access_to_Listed_Station> ::= GRANT ACCESS
                                   STATION <Station_name>
                                   TO {<Group_name> | ALL};

<Station_name> ::= <Identifier>
<Group_name>   ::= <Identifier>

```

Revoke Access to Listed Station

Disables working on a station for a specified group or for all users.

Syntax

```

<Revoke_Access_to_Listed_Station> ::= REVOKE ACCESS
                                       STATION <Station_name>
                                       FROM {<Group_name> | ALL};

<Station_name> ::= <Identifier>
<Group_name>   ::= <Identifier>

```

Devices**Create Device**

Creates a new logical device.

Syntax

```

<Create_a_Device > ::= CREATE DEVICE <Device_Name>
                    DIRECTORY <Path_specification>
                    [COMMENT <Comment>]
                    [LEVEL<Device_RAL>,<Device_WAL>]];

<Device_Name> ::= <Identifier>
<Path_specification> ::= <Character_literal>
<Comment> ::= <Character_literal>
<Device_RAL> ::= <Identifier>
<Device_WAL> ::= <Identifier>

```

Drop Device

Deletes the existing logical device.

Syntax

```

<Drop_a_Device> ::= DROP DEVICE <Device_name>;
<Device_Name> ::= <Identifier>

```

Alter Device

Changes the parameters of the existing logical device.

Syntax

```

<Alter_a_Device> ::= ALTER DEVICE <Device_name>

```

```

[ENABLE | DISABLE]
DIRECTORY <Path_specification>
[COMMENT<Comment>]
[LEVEL (<Device_RAL>,<Device_WAL> )];
<Device_Name> ::= <Identifier>
<Path_specification> ::= <Character_literal>
<Comment> ::= <Character_literal>
<Device_RAL> ::= <Identifier>
<Device_WAL> ::= <Identifier>

```

Grant Access to Listed Device

Grant access to a listed device to a specified group or to all users.

Syntax

```

<Grant_access_to_a_Device> ::= GRANT ACCESS
ON DEVICE<Device_Name>
TO {<Group_name> | ALL};
<Device_Name> ::= <Identifier>
<Group_name> ::= <Identifier>

```

Revoke Access to Listed Device

Revoke access to a listed device to a specified group or to all users.

Syntax

```

<Revoke_access_to_a_Device> ::= REVOKE ACCESS
FROM DEVICE<Device_Name>
TO {<Group_name> | ALL};
<Device_Name> ::= <Identifier>
<Group_name> ::= <Identifier>

```

Grant Access on Unlisted Device

Grant access, to all users or to a specified group, to a device not listed in the list of devices.

Syntax

```

<Grant_access_to_an_Unlisted_Device> ::= GRANT ACCESS
ON UNLISTED DEVICE
TO {<Group_name> | ALL};
<Group_name> ::= <Identifier>

```

Revoke Access on Unlisted Device

Revoke access, for all users or for a specified group, to a device not listed in the list of devices.

Syntax

```
< Revoke _access_to_an_Unlisted Device> ::= REVOKE ACCESS
ON UNLISTED DEVICE
FROM {<Group_name> | ALL};

<Group_name> ::= <Identifier>
```

Get Information about Security Tags**Syntax**

```
SECURITY ( { * |<Column_name>} , { 'R' | 'W' | 'G' } )
<Column name>::= <Identifier>
```