Novell®

SQL *Connector*

SQL Grammar Manual

Printing Date: **October 1, 1999**

About This Manual

Purpose of this Manual

This manual describes the SQL grammar (language and syntax) that is used by SQL *Connector*. The SQL grammar is derived from the ANSI SQL-92 standard and is database independent. Thus, the SQL statements can be used with any database on any supported operating system. The SQL *Connector* SQL grammar is transposed into vendor-specific SQL grammar when executed by the SQL *Connector* Data Request Broker.

This manual provides descriptions and examples that are generic for all databases supported by SQL *Connector*. Unless stated as an exception, SQL statements and procedures developed with the material in this document will be portable across all supported databases.

Intended Audience

This document is intended for programmers who will be creating and maintaining applications which use SQL statements to access multiple databases. It provides provides details about SQL syntax, expressions and built in functions.

Structure of this Manual

This manual consists of a chapters which describes the SQL *Connector* SQL Grammar.

Associated Documents

The SQL *Connector* document set contains these manuals:

- SQL *Connector Overview*
- SQL *Connector Installation Guide*
- SQL *Connector Administration Guide*
- SQL *Connector SQL Grammar Manual*
- SQL *Connector ODBC Programmer's Guide*
- SQL *Connector JDBC Programmer's Guide*

Operating System Conventions

When there are differences in commands, examples, or syntax between operating systems, the following abbreviations are used:

Abbreviation	Meaning
NetWare	the Novell NetWare operating system
Windows	the Microsoft Windows 95/98/NT operating systems

Syntax Conventions

Symbol	Meaning
$	The command line prompt at the left margin indicates that the user is at the operating system level.
[ ]	These symbols enclose optional syntax items.
/* */	These symbols enclose comments.
( )	Parentheses shown in syntax definitions are necessary and must appear literally in the command.
|	Choose item to left or to right of vertical bar.
{ }	These symbols enclose syntax items that may occur multiple (one or more) times.
[,...]	These symbols enclose optional syntax items that may occur multiple (zero or more) times

Table of Contents

About This Manual G–3

Table of Contents G–5

1 Statement Reference G–7
General Information G–7
COPYIN G–8
COPYOUT G–13
DELETE G–18
INSERT G–20
SELECT G–22
SELECT – SELECT Clause G–23
SELECT – FROM Clause G–25
SELECT – WHERE Clause G–27
SELECT – GROUP BY Clause G–32
SELECT – HAVING Clause G–33
SELECT – ORDER BY Clause G–34
SELECT – GUIDE Clause G–35
SELECT INTO G–36
UPDATE G–37

2 Expressions G–39
2.1 Introduction G–39
2.1.1 Character Strings G–39
2.1.2 Numbers G–40
2.2 Constants G–40
2.2.1 Date/Time Constants G–40
2.2.2 Input Formats G–42
2.3 SQL Aggregates G–43
2.3.1 COUNT G–43
2.3.2 SUM G–44
2.3.3 AVG G–44
2.3.4 MIN and MAX G–45
2.3.5 Aggregates and Subqueries G–45
2.3.6 Aggregates and Null Values G–45
2.4 SQL Functions G–46
2.4.1 Numeric Functions G–46
2.4.2 String Functions G–47
2.4.3 Datatype Conversion Functions G–49
2.4.4 Additional String Scalar Functions G–49
2.4.5 Additional Numeric Scalar Functions G–51
2.4.6 Time and Date Functions G–52
2.5 Outer Joins G–53
2.5.1 ODBC Outerjoin Syntax G–53
2.5.2 ODBC Extensions G–54
2.6 SELECT List CASE Extension G–54

3 Environment Variables G–56
3.1 Defining Environment Variables G–56
3.2 Environment Variable Names G–56
3.2.1 SS_FETCH_ISOLATION G–56
3.2.2 SS_ISOLATION G–57
3.2.3 SS_MAXBOUND G–57
3.2.4 SS_MAXFIELDS G–57
3.2.5 SS_MAXTEXT G–57
3.2.6 SS_OJPUSH G–57
3.2.7 SS_TRACE G–58
3.2.8 SS_WAIT G–58
3.2.9 SMARTTRACE G–58

Index G–59

1

Statement Reference

General Information

Table Names and Owner Names

Table names may be in the format `<owner_name>.<table_name>` or just `<table_name>`. When a table is created or imported, the default owner name is the current username (the value of the SQL keyword `USER`). SQL statements that use the table can omit the owner name if the owner name is the same as the current value of `USER`. If an owner name of `"global"` is used when a table is created or imported, then SQL statements that use the table can always omit the owner name, regardless of the value of `USER`.

Case Sensitivity

SQL is *case-insensitive*. You can type SQL commands and statements in any case. Only items enclosed in quotes retain their case. For example, this SELECT statement retrieves records where the TITLE field exactly equals "VP" (not "vp" or "Vp" or "vP"):

```
SELECT * FROM JOB WHERE TITLE = "VP";
```

Note: SQL *Connector* uses tables in physical datatabases. Creating and deleting tables (and indexes) is a physical database administration function and is not supported by SQL *Connector*. Tables in these databases should be created using database administration tools, then imported into SQL *Connector*.

Datatype Support

The table below shows the the datatypes supported by SQL *Connector*. These datatypes conform to the ANSI-92 SQL standard and include SQL *Connector* extensions. Parameters in [ ] are optional..

SQL *Connector* Datatype	Description
Alphanumeric Datatypes	
CHAR(n)	n-length character string
VARCHAR(n)	character string with maximum length of n
Numeric Datatypes	
SMALLINT [*(p,s)*]	2-byte signed integer [*(precision,scale)*]
INT [*(p,s)*]	4-byte signed integer [*(precision,scale)*]
DECIMAL [*(p,s)]*	8-byte signed integer [*(precision,scale)*]
Floating Point Datatypes	
FLOAT [*(n)*]	[*n-length*] 4-byte float
DOUBLE [*(n)*]	[*n-length*] 8-byte float
Date Datatypes	
DATE	date
TIME [(*scale*)]	time
TIMESTAMP [(*scale*)]	date and time

COPYIN

Import Data into Database

Type: SQL Statement

Prompted Syntax

COPYIN *table-name (*|ALL) ;*

or:

COPYIN *table-name*
(column-name [,...]) ;

Transfer Syntax

COPYIN *table-name (*|ALL [format]*
FROM "filespec"
[WITH SUPERSEDE];)

or:

COPYIN *table-name*
(column-name [format] [,...]
FROM "filespec"
[WITH SUPERSEDE];)

Arguments

table-name

The name of the table which is to receive the data.

column-name

The name of the column in *table-name* that is to receive data as defined by *format*, if specified. Nullable columns omitted from the list default to NULL. You cannot omit a non-nullable column from the list of column-names.

format

The input data file format for the column. If *format* is omitted for a column, the data dictionary is used to determine the correct format. The available *format*s are:

Format	Description
I1, I2, I4, I8	The data column has a 1, 2, 4 or 8-byte integer datatype. Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below). Use I8 for binary DATE, TIME, TIMESTAMP and day-time INTERVALs. Use I4 for binary year-month INTERVALs.

Format	Description
F4, F8, FG	The data column has a 4 or 8-byte real, or G_floating format. Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below).
C1...C255	The data column is a fixed length ASCII string of length 1 to 255 bytes or characters. Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below).
C0delim *or* S0delim	The "delim" is the delimiter that terminates the data column, which is a variable length ASCII string. The delimiter is not regarded as part of the data. C0delim reads data until the delimiter is encountered or an end-of-record. S0delim reads data until the delimiter is encountered, even across line feeds. If no delimiter is specified, the column ends at the next comma, tab, or end-of-record. Note that the second symbol of C0delim or S0delim is a "zero" and not the letter "oh". Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below). The complete list of *delim* choices is provided in the Notes section, below.
Q0delim	This format is similar to the S0delim format, except that it accepts quoted strings. Q0delim reads data after the first quote mark (") until it finds another quote mark followed by the delimiter. Q0delim read across line feeds if necessary. To include a quote in the data for a column, use two quotes (""). Additional spaces or tabs outside the quotes will cause errors. A special delimiter of "pc" is provided for the Q0 format. If you specify a format of Q0pc, the delimiter is a comma for all columns except the last column. For the last column, the delimiter is nl (new line). Note that the second symbol of Q0delim is a "zero" and not the letter "oh". The complete list of *delim* choices is provided in the Notes section, below.
X1...X255	A format for a dummy column. COPYIN produces an error if all the bytes of an input record are not accessed. If you only want to import part of each record, use this format to skip a number of bytes in the data file record. Note that a dummy column name must be specified which does not exist in this table. For example, "DUMMY X4" can be used to skip a longword column.
X0delim	A delimiter for a dummy column. The *delim* can be any of the delimiters listed above for C0delim. The specified delimiter will be read and discarded. For example, X0nl causes the rest of the data record to be ignored. A dummy column name must be used with X0delim.

filespec

The name of the ASCII file used for importing into the database.

If you are in Client/Server mode, this file must reside on the Server node.

WITH SUPERSEDE

If this option is present, an existing error file (*tablename*.elg) is overwritten if there are errors in the COPYIN. Normally, if COPYIN encounters errors and the error file already exists, COPYIN will give an error that it cannot overwrite the error file.

Description

COPYIN is used to insert values into a table. The values can come from user input (*prompt mode*) or from ASCII files (*transfer mode*). In both modes, prompt mode and transfer mode, conversion errors and format overflows are detected. In prompt mode, COPYIN places the values you enter directly into the table.

In transfer mode, COPYIN moves records from sequential or fixed length data files to a database. You can also use COPYIN to import data from ordinary ASCII files. COPYIN is intended for technical users. The COPYIN statement provides several unique capabilities:

- If you use the prompted syntax, COPYIN will prompt you for each column value you specify, or you can be prompted for all the columns. If a data conversion error occurs you will be re-prompted for that column. In prompt mode, you can enter numbers and text without quotes. However, you cannot enter keywords (such as TODAY) or input formats such as ^Q.... Only tables can be updated in prompt mode.
- Transaction management is enforced during a COPYIN. If you press Fc (interrupt key), all COPYIN entries will be rolled back. If you press Fz on (EOF key), all the records you entered during the transaction will be committed to the database.
- The ALL feature (or its synonym, "*") can be used to copy data from files with full binary support to a database, even if the file has some datatype formats that are incompatible with the destination database, as long as the proper conversion formats are specified.
- Errors caused by datatype inconsistencies are detected. In prompt mode, you are prompted to re-enter the data. In transfer mode, records that cause errors are written to a new file for editing and re-entry. The error log file is called *tablename*`.elg`.

Notes

The *delim* choices for the C0delim or S0delim format are:

Delimiter	Column Ends at Next Occurrence of
nl	<RETURN>, <CR> or <FF> (use lowercase "L", not "one")
tab	<TAB> or ^I (tab character)
sp	" " (space)
nul	"" (ASCII "0" (zero))
null	"" (ASCII "0" (zero))
comma	,
colon	:
dash	-
lparen	(
rparen	)
bar	|
dot	.
percent	%
star	*
quote	"
apostrophe	'

Delimiter	Column Ends at Next Occurrence of
slash	/
bang	!
at	@
pound	#
plus	+
semi	;
question	?
dollar	$
under	_
tilde	~
grave	‘
greater	>
less	<
ampersand	
equals	=
back	\
caret	^
lsquare	[
rsquare	]
lcurly	{
rcurly	}
c	any single character, except "!", "\", ";"

The COPYIN statement can rapidly append records to database tables if the data file format is the same as the database table format. The file syntax allows binary transfer of data in these situations. The actual speed of the COPYIN statement depends on the byte size, number of columns and datatypes per record, and of course the operating system conditions.

As a general rule it is better to create a table, perform the bulk load (COPYIN), and then create the indices. If you create the indices before performing the bulk load, the transfer will be substantially slower since the index structure must be updated one record at a time.

Data conversion errors are sent to an output file whose name is the same as the input table. The file extension for this error log is `.elg`. This file will have a binary format. A single error record is produced for each input record that produces an input conversion error. Note that when all the bytes of an input record are not accessed, an error is produced. If a subset of each input record is desired, use the "dummy column" formats.

You can import a file containing any of the date datatypes in string format using the C*nnn* or C0*delim* format, but you must use one of the formats accepted for date/time constants *without* the datatype keywords and the quotes. See the "Date/Time Constants" section for a description of these formats.

To import a file that contains scaled integers as ASCII text (e.g., 123.45), you must use character formats. For example, use "`COPYIN SCALE_TABLE (* C0` FROM "DUMMY.DAT";)".

Nullable columns have an indicator byte prior to the data in the data file. The indicator is the ASCII character '0' or '1' if the format is any form of NC; otherwise the indicator is a binary 0 or 1.

To import a nullable column, place an "N" before the desired format (e.g., NF4, NC1...NC255, NC0delim). If you use a format with the "N" prefix, COPYIN will check the indicator byte.

If the indicator byte is even, a null value will be imported and the data portion (which must be included in the data file) will be ignored. If the indicator byte is odd, the data portion that follows the indicator is imported. If you are importing data from an ordinary ASCII file, you can precede null values by a zero, and non-null values by a 1; the data portion of the column must be present, but it can have any value.

Null values cannot be imported to non-nullable columns (but, you can use the X1 format to ignore the indicator byte). If a null format is used to move values to a non-nullable column, records containing null values will be considered conversion errors, and will be written to the error log.

Note: You can't omit non-nullable columns from the COPYIN statement.

Examples

1.
```
*) COPYOUT WRAP_TAB (ID C0null, DESC1 C0nl, DESC2 C0nl,
*)   DESC3 C0null) TO "WRAPOUT.DAT";
*) COPYIN NEW_TABLE (ID S0null, DESC S0null)
*)   FROM "WRAPOUT.DAT";
```

   The above example shows how to concatenate multiple text columns into one, longer contiguous column with embedded line feeds.

   WRAP_TAB is a table that has four columns: an ID column, and three description columns (DESC1, DESC2, DESC3) that are word-wrapped. We want to move them to a table that has only two columns: an ID column, and a DESC column that consists of the former columns DESC1, DESC2, and DESC3. To do this, the data is copied out (with COPYOUT), putting a NULL delimiter where we want each column to end, and an embedded line feed character (NL) in between the word-wrapped columns.

   When the data is copied in (with COPYIN), the S0null format is used to determine the end of the columns, and the embedded line feed characters remain intact within the new, longer DESC column.

2.
```
COPYIN JOB (* Q0PC) FROM "JOB.ALL";
COPYIN DEPT (DEPTNO Q0comma, NAME Q0nl)
     FROM "DEPT.LIS";
```

   The first statement imports data from a file that contains comma-separated quoted strings. The file could look like this:

```
"1","SECRETARY I    ","6504","8994"
"2","SECRETARY II   ","6504","10996"
"3","MGMT TRAINEE   ","8000","14980"
"4","ADMIN ASSISTANT","10041","18940"
"5","PROGRAMMER     ","15002","21999"
"6","DIV MGR        ","18019","34938"
"7","REGIONAL HEAD  ","22104","49942"
"8","VP             ","35161","99536"
"9","EXEC VP        ","50208","499138"
```

   The second statement imports data from a similar file that contains data for only two of the columns in the table.

COPYOUT

Export Data from Database

Type: SQL Statement

Syntax

COPYOUT *table-name (*|ALL [format]*
[TO "filespec"]
[WITH SUPERSEDE];)

or:

COPYOUT *table-name*
(field-name [format] [,...]
[TO "filespec"]
[WITH SUPERSEDE];)

Arguments

table-name

The name of the table that will be exported.

field-name

The name of the field in *table-name* from which data is to be exported. Instead of listing fields, you can use "*" or "ALL" to export all the fields in the table.

format

The output file format for the record or for the particular field if field names are given explicitly. If no *format* is given for a field, the data dictionary is used to determine the correct format. If "*" or "ALL" is used to export all the fields, *format* must be either blank or C0delim. The available *format*s are:

Format	Description
I1,I2,I4,I8	The data column has a 1, 2, 4 or 8-byte integer datatype. Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below). Use I8 for binary DATE, TIME, TIMESTAMP and day-time INTERVALs. Use I4 for binary year-month INTERVALs.
F4, F8, FG	The data column has a 4 or 8-byte real, or G_floating format. Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below).
C1...C255	The data column is a fixed length ASCII string of length 1 to 255 bytes or characters. Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below).

Format	Description
C0delim *or* S0delim	The "delim" is the delimiter that terminates the data column, which is a variable length ASCII string. The delimiter is not regarded as part of the data. C0delim reads data until the delimiter is encountered or an end-of-record. S0delim reads data until the delimiter is encountered, even across line feeds. If no delimiter is specified, the column ends at the next comma, tab, or end-of-record. Note that the second symbol of C0delim or S0delim is a "zero" and not the letter "oh". Precede this format with an "N" if the column is preceded by a null indicator byte in the source file (See Notes, below). The complete list of *delim* choices is provided in the Notes section, below.
Q0delim	This format is similar to the S0delim format, except that it accepts quoted strings. Q0delim reads data after the first quote mark (") until it finds another quote mark followed by the delimiter. Q0delim read across line feeds if necessary. To include a quote in the data for a column, use two quotes (""). Additional spaces or tabs outside the quotes will cause errors. A special delimiter of "pc" is provided for the Q0 format. If you specify a format of Q0pc, the delimiter is a comma for all columns except the last column. For the last column, the delimiter is nl (new line). Note that the second symbol of Q0delim is a "zero" and not the letter "oh". The complete list of *delim* choices is provided in the Notes section, below.
X1...X255	A format for a dummy column. COPYIN produces an error if all the bytes of an input record are not accessed. If you only want to import part of each record, use this format to skip a number of bytes in the data file record. Note that a dummy column name must be specified which does not exist in this table. For example, "DUMMY X4" can be used to skip a longword column.
X0delim	A delimiter for a dummy column. The *delim* can be any of the delimiters listed above for C0delim. The specified delimiter will be read and discarded. For example, X0nl causes the rest of the data record to be ignored. A dummy column name must be used with X0delim.

filespec

The ASCII file name that receives the exported data. If filespec contains any keyword, it must be surrounded by quotes. If filespec is omitted, the default destination is.

WITH SUPERSEDE

If this option is present, an existing output file (specified by *filespec*) is overwritten. Normally, if the output file already exists, COPYOUT will give an error that it cannot overwrite the output file.

Description

COPYOUT performs a fast bulk transfer from a table to an ASCII file. To maximize the speed of bulk data transfers, qualifications are not allowed in a COPYOUT statement. If you want to export only selected records, use other SQL statements to append the selected records to a temporary table which can be copied out. The COPYOUT statement performs several unique functions:

- Formatting can be used with the COPYOUT statement. In the simplest form of the syntax, all fields can be output in a default binary format or a character format.
- The COPYOUT and COPYIN statements can be used together to quickly dump

(export) and re-load (import) or transfer tables with many fields and many records. COPYOUT transfers data just as efficiently as a hard-coded host program.

Notes

The list of *delim* choices for the C0delim or S0delim formats is:

Delimiter	Column Ends at Next Occurrence of
nl	<RETURN>, <CR> or <FF> (use lowercase "L", not "one")
tab	<TAB> or ^I (tab character)
sp	" " (space)
nul	"" (ASCII "0" (zero))
null	"" (ASCII "0" (zero))
comma	,
colon	:
dash	-
lparen	(
rparen	)
bar	|
dot	.
percent	%
star	*
quote	"
apostrophe	’
slash	/
bang	!
at	@
pound	#
plus	+
semi	;
question	?
dollar	$
under	_
tilde	~
grave	‘
greater	>
less	<
ampersand	
equals	=
back	\
caret	^
lsquare	[
rsquare	]
lcurly	{
rcurly	}
c	any single character, except "!", "\", ";"

If you export a table containing any of the date datatypes using the C*nnn* or C0*delim* formats, the dates will be written to the file in the following formats:

Datatype	**Format**
DATE	YYYY-NN-DD
TIME(2)	HH:MM:SS.FF
TIMESTAMP(2)	YYYY-NN-DD HH:MM:SS.FF
INTERVAL YEAR	PYYYYY
INTERVAL YEAR TO MONTH	PYYYYY-NN
INTERVAL MONTH	PNNNNN
INTERVAL DAY	PDDDDD
INTERVAL DAY TO HOUR	PDDDDD HH
INTERVAL DAY TO MINUTE	PDDDDD HH:MM
INTERVAL DAY TO SECOND(2)	PDDDDD HH:MM:SS.FF
INTERVAL HOUR	PHHHHH
INTERVAL HOUR TO MINUTE	PHHHHH:MM
INTERVAL HOUR TO SECOND(2)	PHHHHH:MM:SS.FF
INTERVAL MINUTE	PMMMMM
INTERVAL MINUTE TO SECOND(2)	PMMMMM:SS.FF
INTERVAL SECOND(2)	PSSSSS.FF

If you want to be able to see the scaling in an output file that contains scaled integers, you must use the character formats.

For example, "`COPYOUT SCALE_TABLE (* C0 TO "DUMMY.DAT";)`" writes a file that contains characters. If 123.45 is a scaled value, the character value is also 123.45. If you omit "`C0`", the output file would contain binary numeric values. To re-import the file written by the COPYOUT statement above, execute "`COPYIN SCALE_TABLE (* C0 FROM "DUMMY.DAT";)`".

Nullable fields have an indicator byte prior to the data. The indicator is the ASCII character '0' or '1' if the format is any form of NC; otherwise the indicator is a binary 0 or 1.

To export a nullable field, place an "N" before the desired format (e.g., NF4, NC1...NC255, NC0delim). If you use a format with the "N" prefix, COPYOUT will check the indicator byte.

When copying out a null value (e.g., if the indicator byte is even), the data portion is filled with blanks if the format is NC with a fixed length. There is no data portion at all if the format is NC0delim (the representation of a null is just an ASCII '0' followed by the delimiter). For all other null formats, the data portion is filled with binary 0's.

If the indicator byte is odd (1), the data portion that follows the indicator is exported.

If null values are exported with a non-nullable format, the non-nullable field is set to the default value (0 for a numeric field, blank for a character field). In general,

conversion errors (such as data overflow) do not result in an error message. Instead, the default value (0 for a numeric field, blank for a character field) will be copied out.

Examples

1. `COPYOUT JOB (*) TO "JOB.BIN";`

   This example copies a table to a flat file in binary format.

   Make sure quotes are used if the filespec contains anything other than alphanumeric characters.

2. `COPYOUT JOB (* C0) TO "JOB.ASC";`

   This examples copies a table to a data file in default ASCII format. Note that the ASCII output format places a <TAB> character between each output field as a field delimiter. Thus in general, this approach does not provide uniform columnar listings. This effect can be seen from an editor or by typing \TYPE JOB.ASC in SQL. The <TAB>'s are automatically removed by the COPYIN operation as long as the same format is given. If you just want to produce a ASCII text file with uniform column display, this is better accomplished with the REPORT statement.

3. `COPYOUT DEPT (DEPTNO I4, NAME  C40) TO "DEPTX.DAT";`

   In this case, only two fields are being written out in binary formats, and the formats are being changed. If a table DEPTX is created with this identical format then

   ```
   COPYIN DEPTX (*) FROM "DEPTX.DAT";
   ```

   can be used to perform a bulk load of the new data structure.

4. `COPYOUT JOB(TITLE C4, SALMIN C6, DUMM X0COLON) TO GARB.LIS;`

   In this case the first four characters of the field TITLE will be concatenated with the six digits of SALMIN (left justified into character format, i.e., leading zeros are eliminated). That result is then further concatenated with a colon symbol ":" to end the record. The records are stored in the file GARB.LIS.

5. ```
   COPYOUT JOB (* Q0PC) TO "JOB.ALL";
   COPYOUT DEPT (DEPTNO Q0comma, NAME Q0nl) TO "DEPT.LIS";
   ```

   The first statement exports data to a file in the following format:

   ```
   "1","SECRETARY I    ","6504","8994"
   "2","SECRETARY II   ","6504","10996"
   "3","MGMT TRAINEE   ","8000","14980"
   "4","ADMIN ASSISTANT","10041","18940"
   "5","PROGRAMMER     ","15002","21999"
   "6","DIV MGR        ","18019","34938"
   "7","REGIONAL HEAD  ","22104","49942"
   "8","VP             ","35161","99536"
   "9","EXEC VP        ","50208","499138"
   ```

   The second statement exports two columns from a table to a file in the following format:

   ```
   "1","MARKETING       "
   "2","TRAINING        "
   "3","CREDIT          "
   "4","ACCOUNTS PAYABLE"
     .
     .
     .
   ```
   ```

DELETE

Delete Records

Type: SQL Statement

Syntax

DELETE *FROM table-name*
[USING table-name [corr-name]]
[,...]
[WHERE where-clause]
[GUIDE guide-clause];

Arguments

table-name

The name of the table which is to have records deleted.

USING

The USING keyword is followed by a list of *table-names* separated by commas, or *table-name corr-name* pairs separated by commas. A *corr-name* acts as a synonym for a *table-name* when it is used in the WHERE clause.

where-clause

The complete generality of the SELECT statement where-clause is available to the DELETE statement. See the SELECT statement description for details.

guide-clause

The GUIDE clause to be used in defining a query path for the delete. The syntax supported by *guide-clause* is the same as that described for SELECT. See the SELECT statement description for details.

Description

The DELETE statement is used to delete zero or more records from a single table table-name. If no qualifications are given, all records will be deleted. An SQL extension is provided (the USING clause) which eliminates the ANSI-SQL restriction of only being able to delete based upon fields of the target table-name.

Notes

The USING Clause Extension

The USING clause ANSI-SQL extension allows multiple tables with joins to be used in the WHERE-clause for the DELETE statement.

- Views - You cannot delete records from a view if it is created with any qualifications or with the DISTINCT keyword.

- For cautious users, a good technique to apply when performing deletes from the query language is to first begin a transaction. If unexpected behavior is encountered, then rollback the results. It must be emphasized that if this qualification is omitted, all records in the table really will be deleted. If transaction management is not used as an insurance policy, the only way to recover tables is from backups (assuming that backups are regularly made in your facility).

Examples

1.
```
DELETE FROM JOB WHERE JOBCLS = 3;
```

   This example shows a single record delete. If JOBCLS is a unique index, you are guaranteed of deleting at most one record.

2.
```
DELETE FROM JOB;
COPYIN JOB(*) FROM "JOB.DAT";
```

   This example shows multiple record deletes. All records of the JOB table are deleted, and then they are reloaded via the COPYIN statement.

3.
```
BEGIN TRANSACTION;
DELETE FROM EMP
WHERE NAME LIKE "A%";
IF SQL_RECORDCOUNT > 20 THEN;
  ROLLBACK TRANSACTION;
ELSE;
  COMMIT;
ENDIF;
```

   This example shows cautious multi-record deletes. The deletes are performed within a transaction and the results of the delete are tested before committing.

4.
```
CREATE TABLE JOBX (JOBCLS INTEGER NOT NULL,
    TITLE CHAR(20) NOT NULL, SALMAX INTEGER);
INSERT INTO JOBX
    SELECT JOBCLS, TITLE, SALMAX FROM JOB;

DELETE FROM JOBX WHERE JOBCLS IN
    (SELECT JOBCLS
     FROM JOB
     WHERE SALMAX > 20000);
```

   This example shows complex deletes using subqueries.

5.
```
DELETE FROM EMP
USING JOB
WHERE EMP.JOBCLS = JOB.JOBCLS
AND JOB.TITLE LIKE 'S%'
GUIDE JOB(JOBCLS);
```

   This example shows a DELETE with a USING clause and a GUIDE clause.

6.
```
DELETE FROM EMP E USING EMP M, DEPT D
WHERE E.DEPTNO = D.DEPTNO AND D.MNGRNO = M.EMPNO AND M.NAME LIKE 'ELL%';
```

   This example shows a DELETE with a self-join. In this example, the query defines deletion of all employee records where the department managers name starts with "ELL".

INSERT

Insert Records into the Database

Type: SQL Statement

Syntax

INSERT INTO *table-name*
[(column-list)]
[VALUES (value-list) | select-statement] ;

Arguments

table-name

The name of the table into which the records will be inserted.

column-list

An optional list of field names (separated by commas) which are to have values inserted. The order is arbitrary as long as the value-list matches this order.

Nullable fields omitted from the column-list default to NULL. You can't omit non-nullable fields from the column-list.

value-list

An optional list of values (separated by commas) to be inserted. If the optional column-list is omitted, then the values must be entered in precisely the order that the fields were created ("SHOW FIELDS OF table-name;" will list the actual order). If a field is nullable, the keyword "NULL" may be used to enter a null value in a field.

If the keyword VALUES is used, then select-statement must not be used.

select-statement

A valid SELECT statement can be used to insert records from one table to another table, so long as the column list matches the select field list. The SELECT statement *cannot* contain an ORDER BY clause. If no column-list is given, the destination fields being inserted into must match the source fields in both datatype and order. See the SELECT statement description for details.

Description

This statement inserts one or more records into an existing table. .

When the select-statement option is used to insert records, the operation is called an "insert from a subquery." This generally results in multiple records being inserted into the database. ANSI-standard SQL syntax does not allow expressions in the value-list.

Notes

Datatype conversion is automatic. For character conversions, this includes blank padding and truncation as necessary. For scaled numeric datatypes, decimal values will be rounded as appropriate before storing in the database.

The SQL INSERT statement does not allow expressions as part of the value list.

There are several forms of the INSERT statement:

1. INSERT INTO table-name
   (column-list)
   VALUES (value-list);

In this case, you enter the destination fields in any order desired. The value-list is then presented in the corresponding order.

2. INSERT INTO table-name
   VALUES (value-list);

As long as the value-list contains all the values of the table in the same order as listed by the CREATE TABLE statement, then the column-list can be omitted.

3. INSERT INTO table-name
   SELECT *
   FROM ...;

Here it is assumed that the source and destination tables have the same field names, datatypes and sizes.

SQL *Connector* eliminates the ANSI-SQL restriction against referencing the target table in the subquery. In addition, outer joins are allowed in the subquery.

Aside from the restrictions described here, the select statement functions as described for the SELECT statement documentation.

Inserting Date Values

For INSERT statements with a "VALUES" list, use any format accepted as a date/time constant *including* the datatype keywords and quotes.

Alternatively, input formats can be specified for character date formats:

```
insert into sal (empno, salary, increase, sdate, psalary, pincrease, pdate)
values (12345, 25000, 3000, ^q"NN/DD/YY"("01/02/80"), 23000, 2000,
        ^q"NN/DD/YY"("01/15/79"));
```

SELECT

Query a Database

Type: SQL Statement

Syntax

SELECT *select-clause*
FROM from-clause
[WHERE where-clause]
[GROUP BY group-clause]
[HAVING having-clause]
[ORDER BY order-clause]
[GUIDE guide-clause];

Description

The SELECT statement is the most important statement in the ANSI-standard SQL structure. It allows for very general approaches to qualified record retrieval from a database.

The presentation is given on the following pages with subsections allocated to each of the above clauses. This approach is also logical in terms of simple-to- complex expression usage.

A number of examples are presented for each different clause and where appropriate for specific database types.

Notes

The clauses of the SELECT statement are explained in order and in detail on the following pages. Briefly, the SELECT clause names a list of columns or expressions, the FROM clause a list of tables, the WHERE clause sets conditions on the rows, the GROUP BY clause groups rows together, the HAVING clause sets conditions on the groups, ORDER BY orders the rows, and GUIDE allows the expert user to guide the query path. The SELECT INTO option creates a new table with specified fields, and then updates the new table based upon the rest of the select statement.

When you list more than one table in the FROM clause, SQL behaves as though it is creating a composite table where the rows of the new table are constructed by taking all the possible combinations of rows from all the tables listed in the FROM clause. If there is a WHERE clause, SQL eliminates from this new table all rows that do not meet the conditions of the WHERE clause. All join conditions are specified through the WHERE clause. This modified table is returned to the SELECT clause, where all columns not listed are eliminated. The resulting table is what the SELECT statement returns.

SELECT – SELECT Clause

Specify the Data to Extract From One or More Tables in a Database

Syntax

SELECT [DISTINCT | UNIQUE] *[* | select-list]*

Arguments

DISTINCT

is a keyword that causes SQL to eliminate duplicate rows from the query results. DISTINCT can be used in conjunction with the INTO keyword.

UNIQUE

is a keyword synonym for DISTINCT.

asterisk (*)

is a keyword that causes SQL to select all rows that satisfy the WHERE clause, without eliminating duplicates.

select-list

is a list of column names, constants, and/or expressions separated by commas. A column name or correlation must be unambiguous; use its table name as a prefix if there can be confusion. A column name can take any of the following forms:

- field-name
- table-name.field-name
- correlation-name.field-name
- *
- table-name.*
- correlation-name.*

Notes

If the SELECT statement does not include a WHERE clause, every row will be returned.

The keyword DISTINCT can be used only once in a query.

The asterisk (*) may be used in the select-list to select all columns from all the tables in the FROM clause. You can produce the same result by listing every column name in the select-list. To insure against ambiguity in multi-table join queries, the field being counted should be explicitly stated.

To select all the columns from a single table, use the notation table-name.*, or correlation-name.*

If you specify an aggregate function and a column in the select-list, the column must be used in the GROUP BY list (see GROUP BY for further explanation).

SELECT queries which result in a single row are called "Singleton-select" queries. The most common occurrences are based upon equality qualifications where a unique index exists, and from aggregate selection in the select list.

Examples

1. ```
   SELECT DISTINCT SALMIN FROM JOB;
   ```

   A SELECT clause using the DISTINCT keyword.

2. ```
   SELECT * FROM JOB, EMP WHERE JOB.JOBCLS = EMP.JOBCLS AND EMP.NAME LIKE "E%";
   ```

   A SELECT clause using the * keyword.

3. ```
   SELECT SALMIN FROM JOB;
   SELECT DISTINCT SALMIN FROM JOB;
   SELECT JOB.SALMIN FROM JOB;
   SELECT J.SALMIN FROM JOB J;
   ```

   These examples show various forms of column names.

4. ```
   SELECT * FROM JOB;
   SELECT DISTINCT * FROM JOB;
   SELECT JOB.* FROM JOB;
   SELECT J.* FROM JOB J;
   ```

   These examples show various forms of column names using the * keyword.

5. ```
   SELECT JOBCLS, 1, "Hi", TODAY ^Q"NNYY"("1286"), 3*JOBCLS, STRING (5,
   JOBCLS), COUNT (*) FROM JOB WHERE TITLE = "VP" GROUP BY JOBCLS;
   ```

   Each of the allowable *select-list* elements is used in the above SELECT statement: a field-name, a numeric constant, a string constant, a date/time constant, an input format, an expression, a function, and an aggregate.
   ```

SELECT – FROM Clause

Syntax

FROM *table-name [correlation-name]*
[(option, ...]
[,...])

Arguments

table-name

This is the name of a table or view. A maximum of 16 tables may be used in a SELECT clause.

correlation-name

This is a name which represents the table. The correlation name is of primary use in two situations:

- Long table names exist and you want to use the minimum number of keystrokes in a complex multi-table select statement.
- A table is to be joined to itself (called a self join).

option

Consists of either or both keywords and MINLOCKS in arbitrary order separated by a comma. The option(s) must be surrounded by parentheses.

- **OUTERJOIN** causes the associated table to be "outer joined" to the other tables in the query. Fields from an outer joined table will be null where no matching of the join variable exists. See “Outer Joins” on page 53. for more information on outerjoins.
- **MINLOCKS** determines the level of access other users can have to the table. Note that MINLOCKS is enabled only when the ISOLATION switch is set to NONE and the FETCH_ISOLATION switch is set to NONE or LOCKON. Using the MINLOCKS option in a multi-user environment reduces the probability of locking delays during updates. If you want greater control over file access, you can use the RESERVING-clause of the BEGIN statement.

If MINLOCKS is included, other users are allowed to read and write to the data file. If MINLOCKS is omitted, other users can only read the data file. Thus, it is the reader's responsibility to use MINLOCKS if at least one writer is to be permitted. The possible situations are as follows:

Users		**Consequences**
Reader	Writer	
MINLOCKS MINLOCKS	MINLOCKS no	Both users have access. The reader may see uncommitted data entered by the writer. (Level 1 consistency)

Users		Consequences
no no	MINLOCKS no	Only the first user to request access will get it. The other user will be denied access and will get an error message. Thus, the user who gets access will never read uncommitted data entered by other users. (Level 3 consistency)
Writer 1	Writer 2	
MINLOCKS	MINLOCKS	Both users will have access, and both will be able to read uncommitted data entered by the other. (Level 1 consistency)
MINLOCKS no no	no MINLOCKS no	The first user to request access will get it. The second user will be denied access and will get an error message. Thus, the user who gets access will never read uncommitted data entered by other users. (Level 3 consistency)

Examples

1.
```
SELECT * FROM JOB;
SELECT * FROM JOB, DEPT;
```

   In the latter case, all combinations of records from the JOB and DEPT table will be displayed since no join condition exists (using the demo database, there will be a total of 9x25=225 records).

2.
```
SELECT E.NAME, D.NAME FROM EMPLOYEE E, DEPARTMENT D WHERE E.DEPTNO =
D.DEPTNO AND E.EMPNO < 30;
```

   This example shows the use of correlation names. In instances where the table names are long, or where many fields are selected from several tables, this syntax can result in a considerable saving of keystrokes. Without correlation names, the above SELECT statement looks like:

```
SELECT EMPLOYEE.NAME, DEPARTMENT.NAME FROM EMPLOYEE, DEPARTMENT WHERE
EMPLOYEE.DEPTNO = DEPARTMENT.DEPTNO AND EMPLOYEE.EMPNO < 30;
```

3.
```
SELECT DEPT.NAME, EMP.NAME FROM DEPT, EMP (OUTERJOIN) WHERE DEPT.MNGRNO =
EMP.EMPNO;
```

   This example locates the names of departments and managers, matching them up by MNGRNO (DEPT table) and EMPNO (EMP table). In cases where the DEPT record has an invalid MNGRNO, i.e., there is no employee with a matching EMPNO, the department name is displayed anyway. Without the OUTERJOIN clause, those records would not be seen.

   Note that the OUTERJOIN and MINLOCKS options are not mutually exclusive. The same query could include MINLOCKS and correlation names:

```
SELECT D.NAME, E.NAME FROM DEPT D (MINLOCKS), EMP E (OUTERJOIN,
MINLOCKS) WHERE D.MNGRNO = E.EMPNO;
```

SELECT – WHERE Clause

Specify Search Criteria and Join Conditions

Syntax

WHERE *where-clause*

Arguments

where-clause

is a collection of one or more search conditions connected by the logical operators AND, OR, or NOT. A search condition can be any of the following:

- Comparison Condition
- Join Condition
- Condition with Subquery

Description

Comparison Condition

A comparison condition can have one of the following forms:

- expression comparison-operator expression
- fieldname IS [NOT] NULL
- expression [NOT] BETWEEN expression AND expression
- expression [NOT] IN (item-list)
- column-name [NOT] LIKE "string" [ESCAPE "escape-character"]

  This form of the comparison condition allows pattern matching characters within "string". Pattern matching is discussed below.
- expression AND expression
- expression OR expression

Pattern Matching

- A search condition is successful when the value of the column on the left matches the pattern specified by "string". You can use these wildcard characters in place of other characters in "string":

Character	Match
%	matches 0 or more characters
_	matches a single character
[	begins a group of characters, any one of which can be matched by a single data character.
]	ends the group of characters.

- The NOT option makes the search condition successful when the column on the left does not match the pattern specified by "string".
- If the ESCAPE keyword is used and the escape character is encountered while parsing "string", the escape character is discarded and the character following it is not treated as a wildcard character. With judicious use of the escape character, it is possible to mix wildcard characters in a string with their actual value.

This feature is for wildcards only. It cannot be used to insert quotation marks in a string. If you want to do that, use apostrophes as the string enclosures instead of quotation marks. Or, you can put two quotation marks together to signify one quotation mark and not a terminator.

Examples of each of these comparison condition forms are given in the Examples section, below.

Join Conditions

You join two tables when you create a WHERE clause that links at least one column from one table with at least one column from another. In effect, the join creates a temporary composite table in which each pair of rows (one from each table) that satisfies the join condition is linked together to form a single row.

The structure of a join between two tables (table1 and table2) is:

```
SELECT select-clause
FROM table1, table2 ,...
WHERE table1.field comparison-operator table2.field
```

1. When columns from different tables have the same name, you must distinguish them by prefixing them with a table identifier (name or correlation) and a period, i.e. (`table-name.field-name or correlation-name.field-name`).
2. A multiple-table join is a join of more than two tables. Its structure is similar to that shown previously, except that you must list more than one pair of tables in the FROM clause.
3. A maximum of 16 tables can be joined.
4. You can also join a table to itself in a self-join. To do so, list the table name twice in the FROM clause, assigning it two different correlation names. Use the correlation names to refer to the "two" tables in the WHERE clause.
5. A null value in a field does not match any other value, not even another null. For example, if the DEPTNO field is nullable in both the DEPT and EMP tables, and both tables have a record with a null DEPTNO, then the qualifier EMP.DEPTNO = DEPT.DEPTNO will not match the records with a null DEPTNO. To match these records, you could use the following condition:

   ```
   EMP.DEPTNO IS NULL AND DEPT.DEPTNO IS NULL
   ```

Examples showing each of the join condition forms are given in the Examples section, below.

Subqueries

SELECT statements within WHERE clauses are called subqueries. A subquery may return a single value, a set of values, or no values. The search condition in a SELECT statement can also

- Compare an expression to the result of another SELECT statement with the form:

  ```
  WHERE expression comparison-operator [ALL | ANY] (select-statement)
  ```

- Determine whether an expression is included in the results of another SELECT statement with the form:

  ```
  WHERE expression [NOT] IN (select-statement)
  ```

- Determine whether there are any rows selected by another SELECT statement with the form:

  ```
  WHERE [NOT] EXISTS (select-statement)
  ```

ALL is a keyword that denotes that the subquery may return zero, one, or more values and that the search condition is true if the comparison is true for each of the values returned. If the subquery returns no value, the search condition is true.

ANY is a keyword that denotes that the subquery may return zero, one, or more values and that the search condition is true if the comparison is true for at least one of the values returned. If the subquery returns no value, the search condition is false.

IN is a keyword that asks whether the expression is among the values returned by the following select-statement.

EXISTS is a keyword that asks whether there are any rows returned by the following select-statement. The search condition is true if the subquery returns one or more rows.

NOT is an optional keyword that reverses the truth value of the search condition.

Notes on Subqueries

The following items describe special conditions in the use of subqueries:

1. A subquery may not have more than a single column or expression in its select-list and must not contain an ORDER BY clause.
2. The select-statement may not appear on the left side of a comparison operator, only on the right side. The ANSI standard for SQL does not provide for select statements appearing to the left of the comparison operator.
3. The keywords ANY and ALL may be omitted in a comparison if you know the subquery will return exactly one value. In this case the search condition is true if the comparison is true for the expression and the value returned by the subquery. SQL will display an error if the subquery returns more than a single value.
4. expression IN (select-statement) is equivalent to expression = ANY (select-statement).
5. expression NOT IN (select-statement) is equivalent to expression !=ALL (select-statement).
6. In "A Guide to DB2", Addison-Wesley, 1984, C.J. Date recommends that ANY and ALL never be used with comparison operators such as ANY, =ALL. He states that the EXISTS keyword should always be used in these cases to insure a correct query (the book contains many simple to very complex examples of these cases). Due to the fact that proper use of ALL, ANY and EXISTS requires a strong academic understanding of the field of formal logic, the interested reader is referred to Date. In addition, SQL *Connector* supports correlated subqueries, which are also examined in Date's book. SQL *Connector* supports these constructs with multiple level nesting.

Examples showing the use of subqueries are given in the Examples section, below.

Examples

1. `SELECT * FROM JOB WHERE JOBCLS > 4;`

   This example shows a comparison with the comparison-operator (>).

2. `SELECT * FROM JOB WHERE NOT (JOBCLS > 4);`

   This example shows a comparison with the `NOT` keyword.

3. `SELECT A.JOBCLS, B.JOBCLS FROM JOB A, JOB B WHERE A.JOBCLS > B.JOBCLS + 6;`

   This example shows a self-join comparison with an expression (`B.JOBCLS + 6`).

4. `SELECT * FROM EMP WHERE DEPTNO IS NULL;`

   This example shows the use of the `NULL` keyword.

5. `SELECT * FROM JOB WHERE JOBCLS BETWEEN 3 AND 6;`

   This example shows the use of `BETWEEN`.

6. `SELECT * FROM JOB WHERE JOBCLS NOT BETWEEN 3 AND 6;`

   This example shows the use of `NOT BETWEEN`.

7. `SELECT * FROM JOB WHERE NOT (JOBCLS NOT BETWEEN 3 AND 6);`

   This example shows the use of `NOT` with another comparison.

8. `SELECT * FROM JOB WHERE JOBCLS IN (1,3,5,7,9);`

   This example shows the use of the `IN` keyword.

9. `SELECT * FROM JOB WHERE JOBCLS NOT IN (1,3,5,7,9);`

   This example shows the use of `NOT IN`.

10. `SELECT * FROM DEPT WHERE NAME IN ("BANKING", "TRADING");`

    This example shows the use of `IN` with strings.

11. `SELECT * FROM DEPT WHERE NAME LIKE "%S%";`

    This example shows the use of the `LIKE` keyword and pattern matching. It finds all department names containing the letter "S", anywhere in the name.

12. `SELECT * FROM DEPT WHERE NAME NOT LIKE "%S%";`

    This example shows the use of `NOT LIKE` and pattern matching, finding all records where the NAME doesn't contain an "S".

13. `SELECT * FROM EMP WHERE NAME LIKE "E_LI%";`

    Another pattern matching example is shown here. It finds all employee names where the name begins with E, where the second letter can be anything, the third and fourth letters are L and I, respectively, and then any other letters.

14. `SELECT * FROM EMP WHERE SSNO LIKE "_ _ _\- _ _\- _ _ _ _" ESCAPE "\";`

    This example finds all social security numbers that are 11 characters long and have hyphens in the fourth and seventh positions. (All EMP records will be retrieved.)

15. `SELECT * FROM DEPT WHERE NAME LIKE '%[;%;\;*;_;";]]' ESCAPE ";";`

    This example finds all department names which end in any of the following 6 characters: % \ * _ "] - no records will match these choices.

16. `SELECT * FROM JOB WHERE JOBCLS < 3 OR TITLE LIKE "E%";`

This example finds all records where the job class is less than 3 or where the title starts with the uppercase letter "E".

17. `SELECT EMP.NAME, JOB.TITLE FROM EMP, JOB WHERE EMP.JOBCLS = JOB.JOBCLS AND EMP.EMPNO < 21000;`

    This example shows an equi-join.

18. `SELECT E.NAME, D.NAME FROM EMP E, DEPT D WHERE E.DEPTNO = D.DEPTNO AND E.EMPNO < 21000;`

    This example shows another equi-join.

19. `SELECT E.NAME, J.TITLE, D.DEPTNO FROM EMP E, JOB J, DEPT D WHERE E.JOBCLS = J.JOBCLS AND E.DEPTNO = D.DEPTNO AND E.EMPNO < 21000;`

    This example shows another equi-join.

20. `SELECT A.JOBCLS, B.JOBCLS FROM JOB A, JOB B WHERE A.JOBCLS > B.JOBCLS + 6;`

    This example shows a greater-than-join with a self-join. Note that, in a self-join, the table name must be listed twice in the FROM clause with two different correlation names.

21. `SELECT DISTINCT TITLE FROM JOB WHERE JOBCLS IN (SELECT JOBCLS FROM EMP WHERE NAME LIKE "ELLIS%");`

    This example lists the job titles without repetition for all employees whose name begins with ELLIS.

    The subquery is evaluated and then the valid resultant job classes are used to evaluate the main select expression of the form WHERE JOBCLS IN (...).

    Note that in this case (not generally true), you can also write the query as a two table join:

    ```
    SELECT DISTINCT J.TITLE FROM JOB J, EMP E WHERE J.JOBCLS = E.JOBCLS AND
    E.NAME LIKE "ELLIS%";
    ```

SQL *Connector* was used to test one of Date's examples using the EXISTS keyword. The query is paraphrased as follows. Select supplier names for suppliers such that there does not exist a part that they do not supply. The query with results matching Date's example is shown below:

```
SELECT SNAME FROM S WHERE NOT EXISTS
  (SELECT * FROM P WHERE NOT EXISTS
     (SELECT * FROM SP WHERE SNUM = S.SNUM AND PNUM = P.PNUM));
```

SELECT – GROUP BY Clause

Produce a Single Row of Results for a Set of Rows

Syntax

GROUP BY *group-clause*

Arguments

group-clause

is a column name or a list of column names separated by commas that determines the group. The query result contains a single row for each set of rows that satisfied the WHERE clause and contains a unique value or set of values in the column or columns indicated by group-clause.

Notes

The ANSI-standard definition of the GROUP BY clause restricts what you can enter in the SELECT clause. The select-list may include backend aggregates for any column and/or the name of any column that you also list in the GROUP BY clause. You may not, however, list any column in the select-list that you do not also list in the GROUP BY clause except aggregates. The aggregate argument name does not need to be in the GROUP BY list.

Examples

1. `SELECT D.NAME, COUNT(E.EMPNO) "Emp Count" FROM EMP E, DEPT D WHERE D.DEPTNO = E.DEPTNO AND E.EMPNO < 21000 GROUP BY D.NAME;`

   In this example, we want to list the number of employees in each department beside its department name. We also want to only consider employees with employee numbers less than `21000`. If you want to also list the department number corresponding to the department name, it must be added to the select list and the `GROUP BY` list. Note that `E.EMPNO` does not need to be in the `GROUP BY` list since it is the argument of the count aggregate and it is tied to `D.NAME` through the join clause.

SELECT – HAVING Clause

Apply Qualifying Conditions to Groups

Syntax

HAVING *having-clause*

Arguments

having-clause

is a condition or several conditions connected by ANDs or ORs.

Notes

- Each condition compares one aggregate property of the group either with another aggregate property of the group or with a constant.
- The HAVING clause generally complements a GROUP BY clause. If you use HAVING without GROUP BY, the HAVING clause applies to all rows that satisfy the WHERE clause. Without a GROUP BY clause, all rows that satisfy the WHERE clause make up a single group.
- In multi-table query, you should explicitly define what is being counted when the count aggregate is used to insure against ambiguity.

Examples

The same example as described for the group by clause is extended here to include only the resulting records where the grouping results in a count of 4 or more employees per department.

```
1. SELECT D.NAME, COUNT(E.EMPNO) "Emp Count" FROM EMP E, DEPT D WHERE D.DEPTNO
   = E.DEPTNO AND E.EMPNO < 21000 GROUP BY D.NAME HAVING COUNT (E.EMPNO) > 3;
```

SELECT – ORDER BY Clause

Sort Query Results

Syntax

ORDER BY *column-name [ASC|DESC] [,...]*

Arguments

column-name

is the name of a column by which you want to sort the query results. By default, items are sorted in ascending order.

ASC

is a keyword that specifies that the results should be in ascending order. ASC is the default.

DESC

is a keyword that specifies that the results should be in descending order.

Notes

In the place of column names, you can enter one or more integers that refer to the position of items in the select-list. In this way, you can ORDER BY expressions and aggregates.

Examples

The GROUP BY and HAVING example is extended here to include ordering of the results. First we want to display the results in decreasing order of employee count per department. Then for identical employee counts, group the departments in ascending (alphabetical) order.

```
SELECT D.NAME, COUNT(E.EMPNO) FROM EMP E,DEPT D WHERE D.DEPTNO = E.DEPTNO
AND E.EMPNO < 21000 GROUP BY D.NAME ORDER BY 2 DESC, D.NAME;

LOANS SOUTH AMERICA           4
MARKETING                     3
ACCOUNTING                    2
ACCOUNTS PAYABLE              2
CUSTOMER SERVICES             2
. . .
SECURITY                      2
FIDUCIARY                     1
TRAINING                      1
```

SELECT – GUIDE Clause

Define a Query Path

Syntax

GUIDE *table-name[(index-number)] [,...] ;*

Arguments

table-name

is the identifier specified in the FROM clause. It can be a table name or a correlation name.

index-number

defines the index number which is to be used for the index.

Description

Use the GUIDE clause to define the query path to be taken in performing a query. The GUIDE clause allows you to override the internal query optimizer. For multiple-table queries with large databases, the performance improvements can often be dramatic.

If you specify the correlation name but omit the index number, you force a sequential search of that table. In some instances, this can actually result in a slightly faster retrieval.

You *must* enumerate leading tables in the GUIDE clause. In other words, you cannot specify an index on the second table in the query path without also specifying the first table. For example, if the query optimizer would normally search the EMP table first and the DEPT table second, and you want to keep this order of tables but just specify an index for DEPT, you must also enumerate EMP in the GUIDE clause:

```
GUIDE EMP(1), DEPT(0)
```

Here, EMP is searched first on index 1 (NAME), and DEPT is searched next on index 0 (DEPTNO). If, instead, you merely specify:

```
GUIDE DEPT(0)
```

the search on the DEPT table will override EMP as the first table in the search.

Examples

1. `GUIDE E(1), D(0)`

   Access the EMP table first using index 1, then access the DEPT table using index 0.

2. `insert into joby select * from job where jobcls > 0 guide job(0);`

   Insert all records from the JOB table in the JOBY table. Note that the use of JOBCLS in the WHERE clause is required since index 0 of the JOB table (JOBCLS) is used in the GUIDE clause.

3. `SELECT E.NAME, D.NAME, D.DEPTNO FROM EMP E, DEPT D WHERE E.DEPTNO = D.DEPTNO AND E.NAME LIKE "ELL%" AND D.DEPTNO > 2 GUIDE D, E(3);`

   Access the DEPT table first with a sequential scan of the data for records where DEPTNO > 2. Then for each of these cases, find the corresponding EMP record by going through the DEPTNO field index.

SELECT INTO

Create a New Table with Existing Records

Type: SQL Statement

Syntax

SELECT INTO *table-name [(field, ...]*
AS select-statement;)

Arguments

INTO

Is a keyword which indicates that a new table is to be created.

table-name

Is the name of the table that is to be created and then populated.

field

Is the name of the fields to include in the table.

select-statement

Is any legitimate select-statement. The select-statement should not contain an ORDER BY clause. If an ORDER BY clause is specified, it will be ignored.

Description

The SELECT INTO structure is an SQL extension which is convenient for prototyping and experimentation where data needs to be modified. A new FILE type table is created and is populated with data based upon the query syntax chosen.

Examples

1. `SELECT INTO JOBX AS SELECT * FROM JOB WHERE JOBCLS < 3;`

   A new table `JOBX` is created with the same fields as the existing table `JOB`. All data from `JOB` where `JOBCLS < 3` are transferred to `JOBX`.

2. `SELECT INTO EMPDEPT (DEPTNO, EMPCNT) AS SELECT DEPT.DEPTNO, COUNT(EMP.NAME) FROM DEPT, EMP WHERE EMP.DEPTNO = DEPT.DEPTNO AND EMP.NAME LIKE "A%" GROUP BY DEPT.DEPTNO HAVING COUNT(EMP.NAME) > 2;`

   This complex statement shows how a new table and new field (whose name is "`EMPCNT`") can be created, based on a multiple table join with `GROUP BY` and `HAVING` clauses.

UPDATE

Modify Records

Type: SQL Statement

Syntax

```
UPDATE      table-name
            [USING table-name [corr-name]]
                [,...]
            SET field = expr
                [,...]
            [WHERE where-clause]
            [GUIDE guide-clause];
```

Arguments

table-name

The name of the table which is to have records modified. The table-name definitions also allow for use of correlation-names (see the examples, and also the SELECT statement description).

USING

The USING keyword is followed by a list of *table-names* separated by commas, or *table-name corr-name* pairs separated by commas. A *corr-name* acts as a synonym for a *table-name* when it is used in the WHERE clause.

field

The field name in table-name which is to be modified by the expression expr.

expr

Any valid SQL expression. The keyword NULL can be used to set the value of a nullable field to null.

where-clause

The WHERE clause to be used in qualifying the update. The generality supported by the where-clause is the same as that described for the SELECT statement (see the SELECT statement description of this Chapter).

guide-clause

The GUIDE clause to be used in defining a query path for the update. The generality supported by *guide-clause* is the same as that described for SELECT. (See SELECT for details.)

Description

This statement is used to update one or more records in the table called table-name. An SQL extension is provided (the USING clause) which eliminates the ANSI-SQL restriction of only being able to update based upon fields of the UPDATE table-name.

Notes

The most widely used forms of UPDATE are:

- Single-record updates - modify one or more fields of a single record.
- Multiple-record updates - modify fields of a set of records.
- Updates with subqueries - this type of update allows for very complex operations based upon one or more tables.
- Multiple-table updates - since relational systems only allow for a single table at a time to be updated, multiple table updates should be included within transactions, to insure against problems of inconsistent results (known as the "referential integrity" problem).

The USING clause extension - This ANSI-SQL extension allows for multiple tables with joins for specifying the fields and where-clause to be used in the update. There are limits that vary depending on the database type.

Examples

1. ```
   UPDATE JOB SET JOBCLS = JOBCLS+10, SALMAX = SALMAX*1.2  WHERE JOBCLS = 3;
   ```

   This example shows a single record update.

2. ```
   UPDATE JOB SET JOBCLS = JOBCLS+20 WHERE TITLE LIKE "S%";
   ```

   This example shows multiple record updates.

3. ```
   UPDATE EMP SET JOBCLS = NULL WHERE JOBCLS = 20;
   ```

   This example shows putting a NULL value in a field.

4. ```
   UPDATE JOBX SET JOBCLS = JOBCLS+20, SALMAX = SALMAX*1.1, SALMIN = SALMIN+100
   WHERE JOBCLS IN (SELECT JOBCLS FROM JOB WHERE SALMAX > 10000);
   ```

   This example shows updates with subqueries.

5. ```
   BEGIN;
   UPDATE FUNDA SET AMT = AMT+2600 WHERE ACCT = 1234;
   UPDATE FUNDB SET AMT = AMT-2600 WHERE ACCT = 1234;
   COMMIT;
   ```

   This example shows multiple table updates.

6. ```
   UPDATE EMP USING JOB SET NAME = ' ' WHERE EMP.JOBCLS = JOB.JOBCLS AND
   JOB.TITLE LIKE 'S%' GUIDE EMP(JOBCLS);
   ```

   This example shows an update with `USING` and `GUIDE`.

   The EMP table is having all employee names that have job titles starting with 'S' set to blank values (not practical, but illustrative).

7. ```
   UPDATE EMP E USING JOB J, SAL S SET NAME = 'SPECIAL_CASE' WHERE E.EMPNO =
   S.EMPNO AND E.JOBCLS = J.JOBCLS AND J.TITLE LIKE 'S%';
   ```

   This example shows a USING clause with correlation names.
   ```

2

Expressions

2.1 Introduction

The following discussion on expressions concerns those evaluated in SQL statements. Expressions can be used in all types of SQL statements unless documented otherwise.

In brief, an expression is a number or character string that can be computed within the database. Expressions can consist of the following elements:

- fields (basic elements defining a table)
- literals (numeric constants or character strings)
- date/time constants
- input date/time formats
- arithmetic operators

In addition, expressions can be built up using arithmetic and string functions and aggregate operators. Parentheses should be used to indicate grouping.

The examples in the previous chapter have used fields, literals, arithmetic operators, and the aggregate function COUNT in SELECT statements. In the following example, SALMIN is a field, ">" is an arithmetic operator, and 60000 is a numeric literal:

```
select * from job where salmin > 60000;
```

2.1.1 Character Strings

A character string must begin with either a quote mark (") or an apostrophe (') and must end with the same character. Examples of valid character string expressions are:

```
"Job Class"
'SECRETARY II'
```

Surround the string with apostrophes if you want to include a quote mark within the string. Or, surround the string with quotes if you want to include an apostrophe within the string, as in the following example:

```
"Don't press the ESCAPE key."
```

In some cases, you may want to include quote marks and apostrophes in the same string. Use two apostrophes ('') to include an apostrophe in a string surrounded by apostrophes. Or, use two quote marks ("") to include one quote mark in a string surrounded by quotes, as shown below:

```
"He said, ""Don't press the ESCAPE key""."
```

The string """" represents a string with a single quote sign as its contents. The string '''' represents a string with a single apostrophe as its contents.

2.1.2 Numbers

Numbers begin with a digit and consist of digits, an optional sign, an optional decimal point followed by optional digits, and an optional exponent. The optional exponent is introduced by a letter "e" or "E", followed by an optional sign (+ or -) and one to four digits. Embedded blanks are not allowed in a number.

The following sections will show how to enhance SELECT statements with date/time constants, input date/time formats, backend aggregates, and backend functions.

2.2 Constants

Constants can be used as expressions in SQL statements. You can use previously defined date/time constants to specify a specific date or time, or you can create your own date/time constants using input formats.

2.2.1 Date/Time Constants

There are two ways to specify constant values for date/time datatypes in SQL. You can use such constants in a WHERE–clause or HAVING–clause, in the SET–clause of an UPDATE statement, or in the VALUES–list of an INSERT statement.

- You can use date and time constants with the following formats in SQL. (Note that you should not use quotes around these date and time constants.)

Format	Example
NN/DD/YY	12/24/93
NN/DD/YYYY	12/24/1993
HH:MM	17:01
HH:MM:SS	17:01:59
NN/DD/YYYY + HH:MM:SS	12/24/1993 + 17:01:59

- You can use a datatype keyword followed by a date/time string in quotes whenever you want to specify a date/time constant.

These are examples of valid date/time constants (in COPYIN statements, you should omit the datatype keywords and the quotes):

Constant	Description
DATE '1992-12-04'	4 Dec 1992
TIME '0'	midnight
TIME '13:45'	1:45 PM (second omitted)
TIME '13:45:22'	1:45:22 PM
TIME '13:45:22.3'	1:45:22.3 PM
TIMESTAMP '1992-12-04 13:45:22'	4 Dec 1992 1:45:22 PM
TIMESTAMP '1992-12-04:13:45:22'	note colon instead of space
INTERVAL '5-03' YEAR TO MONTH	5 years 3 months
INTERVAL -'5' YEAR	negative 5 years
INTERVAL '-5' YEAR	minus sign inside quotes
INTERVAL +'5 15' DAY TO HOUR	positive 5 days 15 hours

Constant	Description
INTERVAL -'5 13:45' DAY TO MINUTE	negative 5 days 13 hours 45 min
INTERVAL '13:45' HOUR TO MINUTE	13 hours 45 minutes
INTERVAL '5 13:45:22.50' DAY TO SECOND(2)	5 days 13 hours 45 minutes 22 1/2 seconds

The quoted string must include the subfields of the datatype in descending order. Leading zeros in TIME constants can be omitted. Years must include 4 digits ("1994", not "94") and hours must be specified according to a 24-hour clock. For portability, use hyphens between "year-month-day", a single space between "day hour", colons between "hour:minute:second", and period between "second.fraction".

Built-in Date/Time Constants

There are a number of predefined constants available for getting the current time, date, or day in different formats. These constants can be used as expressions in SQL statements. The following table shows the constants and describes their output. Examples of their usage in SELECT statements follow the table.

Keyword	Description
Variables:	
TODAY	current date
DAYTIME	current time
NOW	current date and time
MIDNIGHT	date and time of previous midnight
CURRENT_DATE CURRENTDATE	current date
CURRENT_TIME CURRENTTIME	current time, accurate to seconds
CURRENT_TIME(2) CURRENTTIME(2)	current time, accurate to hundredths of seconds
CURRENT_TIMESTAMP CURRENTMOMENT	current date and time, accurate to hundredths of seconds
CURRENT_TIMESTAMP(0)	current date and time, accurate to seconds
VAXTIME	current date and time
VAXMIDNIGHT	date and time of previous midnight
Constants:	
YEAR, YEARS	1 year
MONTH, MONTHS	1 month
WEEK, WEEKS	7 days
DAY, DAYS	1 day
HOUR, HOURS	1 hour
MINUTE, MINUTES	1 minute
SECOND, SECONDS	1.00 second

2.2.2 Input Formats

An input format is used to convert a character string (*not* a database field) to the 4- or 8-byte binary integer code representation in which the date values are stored. In this way, you can compare date values to date fields when selecting records. There are four input date/time format specifiers and one UIC specifier:

Input Specifier	Description
^D	longword date format
^T	longword time format
^S	longword date/time format
^Q	quadword date/time format

The general syntax for specifying date/time is:

specifier["template"]("text")

The default templates are:

Specifier	Default Template
^D	"dD-AAA-YYYY"
^T	"hH:MM:SS.FF"
^S	"dD-AAA-YYYY hH:MM:SS"
^Q	"dD-AAA-YYYY hH:MM:SS"

This expression: `^D("1-Jan-1980")`, has the value 0, since 1-Jan-1980 is chosen as the reference date. And this expression: `^T("0:00:00.00")` is 0 (midnight).

Any value earlier than the reference date results in a negative integer. Dates later than the reference result in positive values.

If you don't want to use the default input templates, you can specify your own. Date templates can be made up of the characters shown in the table below

Symbol	Description
Y	Year (numeric)
R	Year, rounded closest to initial value
Z	Next year
Q	Quarter
N	Month number (numeric)
D	Day (numeric)
J	Julian day (1-Jan = 1)(numeric)
W	Weekday name (alphabetic)
E	Weekday name (allows abbreviation in input string) (alphabetic)
K	Weekday number (Sunday=1)(numeric)
H	Hour (0 to 23) (numeric)
M	Minute (numeric)
S	Second (numeric)
F	Fraction (hundredth) of second

Symbol	Description
A	Month name (alphabetic)
L	Month name (allows abbreviation in input string) (alphabetic)
T	Hour (1 to 12) (numeric)
C	"AM" or "PM" (alphabetic)

The alphabetic symbols may be upper or lower case.

Non-alphabetic characters may be used as delimiters in the template, and must be matched positionally in the input string. For example, the comma (,) in the following example serves as a delimiter:

```
^D"AAA,YYYY"("FEB,1988")
```

The specifiers, "R" and "r" allow you to specify a 2-digit year field, and cause the value to be interpreted as "rounded" to the nearest decade, century or millennium from some initialized value, which will normally be the current system date/time. Thus, if the current year is 1990, a value of 50 in an "RR" string will be interpreted as 1950, because 1950 is closer to 1990 than 2050 is. Similarly, a value of 30 in an "RR" string will be interpreted as 2030, because 2030 is closer to 1990 than 1930 is. A value of 40 in the "RR" string will be rounded up to 2040, even though 1940 and 2040 are the same number of years away from 1990. This allows convenient input of dates past the year 2000.

Weekday fields (W, E, K) may be used to check the validity of an input string. For example, if you were to use the following input format in an SQL statement, you would receive an error because Feb 2, 1988 was not a Wednesday:

```
^Q"AAA-dD(WWW)-YYYY"("FEB-2(WED)-1988")
```

You can use input formats in SELECT statements. The next example finds all records where SDATE is past Oct 1, 1981:

```
select empno, sdate from sal where sdate > ^q"dD-AAA-YYYY"("1-OCT-
1981");
```

2.3 SQL Aggregates

SQL aggregates perform simple statistical functions on the set of rows returned by the SELECT statement. Aggregates can be used in a SELECT clause, WHERE clause or a HAVING clause anywhere a constant may appear.

The following backend aggregates are supported:

1. COUNT – obtains the number of rows selected
2. SUM – obtains the sum of a column for all rows selected
3. AVG – obtains the average of a column for all rows selected
4. MIN – obtains the minimum value in a column for all rows selected
5. MAX – obtains the maximum value in a column for all rows selected

2.3.1 COUNT

To find out the number of records retrieved by a SELECT statement, use the COUNT aggregate. This SELECT statement, for example, gives a count of the employee records where the employee name starts with "A":

```
select count(name) from emp where name like "A%";
```

You can also use the DISTINCT keyword with COUNT to get a count of unique values for a field:

```
select count (distinct deptno) from emp;
```

To get a count of the number of records in each group, use the GROUP BY clause. The following SELECT statement lists all the department names along with the number of employees in each department:

```
select dept.name, count(emp.name) from dept, emp where dept.deptno =
emp.deptno group by dept.name;
```

Warning: If you use a GROUP BY clause, the *only* items the SELECT clause can contain are aggregates and items that are also in the GROUP BY clause.

2.3.2 SUM

The SUM aggregate finds the sum of the values in a column. For example, to find the sum of the SALMIN column of the JOB table, type:

```
select sum(salmin) from job;
```

Or, to find the sum of the SALMIN column for only the jobs with job classifications less than 5:

```
select sum(salmin) from job where jobcls < 5;
```

As with the COUNT aggregate, you can find sums within groups:

```
select title, sum(salmax) from job group by title;
```

The above example finds the sum of the SALMAX column within each job title. However, since there is only one record for each job title, the SUM of SALMAX is the same as just SALMAX in this case.

2.3.3 AVG

With the AVG aggregate, you can find the average of a column of fields. Consider the following example:

```
select avg (salmax) from job;
```

A single value will be returned, which is the average of the SALMAX field for all records in the JOB table.

As with any other aggregate, you can find an AVG of records within a group using the GROUP BY clause:

```
select deptno, avg (jobcls) from emp group by deptno;
```

The above SELECT statement lists the department number and average job classification for the employees in each department.

2.3.3.1 Averages of Intervals

You can get more precision when you average intervals by adding more precise interval with a value of zero to the intervals before you average them. This converts the intervals you are averaging to more precise intervals.

For example, if "ELAPSED_DAYS" has an "INTERVAL DAY" datatype, then `AVG (ELAPSED_DAYS)` has the same datatype. However, `AVG (ELAPSED_DAYS + INTERVAL "0" HOUR)` gives the average with an "INTERVAL DAY TO HOUR".

2.3.3.2 Averages of Absolute Dates/Times:

You can compute averages of DATEs, TIMEs, and TIMESTAMPs by subtracting a base date from each value to convert each date to an interval. After you average the resulting intervals, add the base date back to the interval.

This example finds the mid-point between BILLDATE and SHIPDATE:

```
SHIPDATE + (BILLDATE - SHIPDATE)/2
```

This example finds the average BIRTHDATE in a group of records:

```
CURRENT_DATE + AVG ( (BIRTHDATE - CURRENT_DATE) DAY )
```

This example finds the average estimated time of arrival (ETA) and forces the intermediate interval to be a DAY TO SECOND(2) interval:

```
CURRENT_TIMESTAMP + AVG ((ETA-CURRENT_TIMESTAMP) DAY TO SECOND(2))
```

2.3.4 MIN and MAX

The MIN and MAX aggregates allow you to find the minimum and maximum values in a column. To expand on the previous example, the following SELECT statement will list the department number, average, minimum, and maximum job classification for the employees in each department number:

```
select deptno, avg(jobcls), min(jobcls), max(jobcls) from emp group by
deptno;
```

As a more complex example, you could list the same information *only* for those departments where the average job classification is less than 6:

```
select deptno, avg(jobcls), min(jobcls), max(jobcls) from emp group by
deptno having avg(jobcls) < 6;
```

2.3.5 Aggregates and Subqueries

As stated earlier, aggregates can only be used in the SELECT clause or the HAVING clause, and *not* in the WHERE clause. But, suppose you want to find the job title of the job with the minimum job classification. Intuitively, you may come up with this SELECT statement:

```
select title from job where jobcls = min(jobcls);
```

However, this statement is illegal since there is an aggregate in the WHERE clause. To get around this problem, use a subquery:

```
select title from job where jobcls = (select min(jobcls) from job);
```

In this way, the aggregate, MIN, is in a SELECT clause, where it is allowed.

2.3.6 Aggregates and Null Values

If no records satisfy the WHERE clause or if all the values are NULL, the value of an aggregate will be NULL (even if the database type does not support null values). This indicates that no data is available.

If some of the values that satisfy the WHERE clause are null, those values are not included in computation of an aggregate. Thus, the SUM of 1, 5 and NULL is 6, and the AVG is 3 (not 2).

Arithmetic operations on null values result in a null value. Thus, `1 + 5 + NULL = NULL`.

2.4 SQL Functions

SQL functions perform various mathematical, string, and datatype conversion functions on single fields or values. These functions may be used in SELECT lists, WHERE clauses, and HAVING clauses. The available functions are listed below:

2.4.1 Numeric Functions

2.4.1.1 ABS (<expression>) or { fn ABS (<expression>) }

ABS gives the absolute value of a numeric expression. The numeric expression can consist of binary integers, decimal integers, or floating decimals. An example of a valid expression using ABS is:

```
ABS(salmin - 20000)
```

You could use this expression in a SELECT statement as follows:

```
select title from job where abs(salmin - 20000) > 5000;
```

2.4.1.2 INT2 (<value>) and INT4 (<value>)

INT2 and INT4 are used to convert a binary integer, decimal integer, floating decimal, or character string into 2 and 4-byte integers, respectively. Note that a character string to be converted must contain only digits or be zero-length. If the character string contains invalid characters, the result will be 0 (zero). If the value to be converted is too large for the datatype, the result will be inaccurate.

The following SELECT statement will find all pairs of employees such that one of the employee's job classification is 3 greater than the other:

```
select e1.name, e1.jobcls, e2.name, e2.jobcls from emp1, emp2 where
e1.jobcls > int2(e2.jobcls) + 3;
```

2.4.1.3 FLOAT4 (<value>) and FLOAT8 (<value>)

FLOAT4 and FLOAT8 are used to convert a binary integer, decimal integer, floating decimal, or character string to a 4- or 8-byte floating point number, respectively. Note that a character string to be converted must contain only digits or be zero-length. If the character string contains invalid characters, the result will be 0 (zero). If the value to be converted is too large for the datatype, the result will be inaccurate.

The following example shows a valid use of FLOAT4 - converting the 4-byte integer field SALMIN to a 4-byte floating point:

float4(salmin)

2.4.1.4 MOD (<dividend>,<divisor>) or {fn MOD(<dividend>,<divisor>) }

MOD finds the remainder of one number divided by another. The numbers can be binary integer, decimal integer, or floating point expressions. Note, however, that if a floating point expression is used, you may receive a minor round-off error in the result.

Also, if the result is too large to express easily as an integer (because one of the arguments was an H floating type, for example), the result will be 0.

The following expression would find the remainder of SALMAX/10000:

```
mod(salmax,10000)
```

If SALMAX were 25000, the result of the above expression would be 5000. A more complex example of the MOD function is:

```
select salmax, mod(salmax,10000), salmax - mod(salmax,10000) from job;
```

2.4.2 String Functions

2.4.2.1 CONCAT (<s1>, <s2>) or { fn CONCAT(<s1>, <s2>) }

CONCAT will concatenate two character string expressions, such as:

```
concat(first, last) or {fn concat(first, last)}
```

However, if the first string expression is longer than one character, trailing blanks will be truncated. So, if FIRST is "John " and LAST is "Doe" in the above example, the result of the concatenation would be:

```
JohnDoe
```

With the { fn...} syntax, if either of the strings is null, the result is null.

To insert a space in between the two concatenated fields in the previous example, use a nested CONCAT function:

```
concat(first, concat(" ", last))
```

CONCAT can be used in the select-list of a SELECT statement:

```
select name, concat("Mr. or Mrs.", concat(" ", name)) from emp;
```

or

```
select name, {fn concat("Mr. or Mrs.", {fn concat(" ", name)})} from
emp;
```

2.4.2.2 LOWER (<str>) or { fn LCASE (<str>) }

LOWER and LCASE are used to convert a string to lowercase.

```
select * from job where title = lower(j_title);
```

or

```
select * from job where title = {fn LCASE(j_title)};
```

2.4.2.3 NVL (<expression>, <string literal>)

NVL allows substitution of a specified constant in place of a NULL value. If the first argument is NULL, the second argument (usually a constant of the same datatype as the first argument) is returned. If the first argument is not NULL, the first argument is returned. For example:

```
nvl(title, "N/A")
```

If TITLE is null, the text string "N/A" is returned.

There are two synonyms for the NVL function: ISNULL and IFNULL. Either of the synonyms can be used in place of "NVL".

2.4.2.4 STRING (<length>, <expression>)

STRING converts an <expression> to a string with a specified <length>. For example,

```
string(6, salmax)
```

converts SALMAX to a 6-character string. If <length> is greater than the string version of <expression>, then the length of the result string is equal to the length of <expression>.

In the case of a quadword date field, the date will be converted to a string in a format shown by the following example. If HIREDATE represented the date: 3-17-1989 9:33:45.01, then:

```
string(16, hiredate)
```

would result in the string `1989031709334501`. Using 0 as <length> results in a string of the default length of <expression>. For example,

```
string(0,empno)
```

results in a string of length 11, because that is the length of EMPNO. The string lengths will be determined according to the following table:

An expression of this type...	**will be converted to a string of this length**
INTEGER1	4
INTEGER2	6
INTEGER4	11
Packed Decimal of length n	n
Decimal of length n	n
Character string of length n	n

2.4.2.5 SUBSTRING (<pos>, <len>, <str>) or { fn SUBSTRING (<str>, <pos>, <len>) }

SUBSTRING will result in a portion of a given character string. For example:

```
substring(3, 10, title)
```

or

```
{fn substring (title, 3, 10)}
```

will result in a string 10 characters long that is the 3rd through 12th characters of TITLE. So, if TITLE is "`ACCOUNTING REP`", the result of the above SUBSTRING is "`COUNTING R`".

The next example shows SUBSTRING in a SELECT statement:

```
select name, substring(1, 5, name) from emp;
```

or

```
select name, {fn substring (name, 1, 5)} from emp;
```

The employee's full name, along with the first 5 characters of the employee's name, will be displayed.

If a substring goes beyond the length of the original string, blanks will be inserted at the end of the string so it is the length of the original string. For example:

```
substring(1, 10, empno)
```

will result in a string 10 characters long, even though EMPNO is only 5 characters long. The first 5 characters will contain the EMPNO string, and the last 5 characters will be blanks.

The starting point and length you specify in SUBSTRING (1 and 10, respectively, in the above example) must be between 1 and 255.

2.4.2.6 UPPER (<str>) or { fn UCASE (<str>) }

UPPER is used to convert a string to uppercase.

```
select * from emp where name > upper(e_name);
```

or

```
select * from emp where name > {fn UCASE(e_name)};
```

where "e_name" is a bound variable string.

2.4.3 Datatype Conversion Functions

2.4.3.1 TO_CHAR (<numeric value>)

TO_CHAR converts a numeric value to a character string. For example: `to_char (empno)`

2.4.3.2 TO_DATE (<string>)

TO_DATE converts a character string that contains a date in the specified format into a date datatype. For example: `to_date("61-NOV-30", "YY-AAA-DD")`

2.4.4 Additional String Scalar Functions

Where appropriate, these functions can be nested.

2.4.4.1 { fn ASCII (<str>) }

ASCII returns the numeric value of the first character in <string> as an unsigned integer. For example:

```
select { fn ASCII(name) } from emp;
```

returns the ASCII code of the first character of name for each record in emp. If the string is NULL, the result is NULL.

Display width: The number of characters necessary to represent the largest character. For single-byte character sets, this is 3; for double byte it is 5.

2.4.4.2 { fn CHAR (<code>) }

CHAR returns the character corresponding to the given ASCII <code>. <Code> should be between 0 and 255. For example,

```
select { fn CHAR (65) } from dept;
```

returns "A" for each record in emp.

2.4.4.3 { fn DATABASE () }

DATABASE returns the character string name of the currently open database.

2.4.4.4 { fn INSERT (<s1>, <pos>, <len>, <s2>) }

INSERT returns a string where <len> characters have been deleted from <s1> beginning at character position <pos> and where <s2> is inserted beginning at character position <pos>. For example:

```
select {fn INSERT(name,5,6,"FOO") } from emp where
name="OBERMILLER_T";
```

will return the string, "OBERFOO_T".

<s2> replaces the deleted characters. Even if the length of <s2> is greater than <len>, only the deleted characters are replaced, making the length of the result string longer than <s1>. For example:

```
select {fn INSERT(name,5,2,"HELLO")} from emp where
name="OBERMILLER_T";
```

will return the string, "OBERHELLOLLER_T". If either <s1> or <s2> is NULL, the resulting string is NULL.

2.4.4.5 { fn LEFT (<str>, <len>) }

LEFT returns the character string of length <length> excerpted from the beginning of <string>. For example:

```
select {fn LEFT(name, 2)} from emp;
```

will return the first two letters of each name in emp. If <length> is greater than the length of <string>, then the entire <string> is returned. If <string> or <length> is NULL the result is NULL. If <length> is 0 the result is a zero-length string. If <length> is negative the result is NULL.

Display width: the minimum of <length> and the width of <string>.

2.4.4.6 { fn LENGTH (<str>) }

LENGTH returns the number of characters in a string after trimming trailing blanks. For example:

```
select {fn LENGTH(name)} from emp where name="OBERMILLER_T";
```

returns the integer 12 as a result. If <string> is NULL, the result is NULL.

Display width: 10 (big enough to display a positive 4-byte integer)

2.4.4.7 { fn LOCATE (<s1>, <s2> [, <pos>]) }

LOCATE returns the starting position of the first occurrence of <s1> within <s2>. If the optional <pos> argument is present, the search begins at that position in <s2>. If <s1> is not found then 0 is returned.

```
select { fn LOCATE ("LAND", name) } from emp where name="MARKLAND_C";
```

returns the integer 5. If any argument is NULL the result is NULL. If <pos> is less than 1, the result is NULL.

Display width: 10 for a nonnegative 4-byte integer.

2.4.4.8 { fn LTRIM (<string>) }

LTRIM returns <string> with leading blanks removed.

If <string> is NULL, LTRIM returns NULL.

2.4.4.9 { fn REPEAT (<string>, <count>) }

REPEAT returns a character string composed of <string> repeated <count> times.

Trailing blanks are not trimmed, therefore the result variable needs to be of accommodating size. Alternatively, RTRIM can be nested to trim trailing blanks. For example:

```
select {fn REPEAT({fn RTRIM(name)},3)} from emp where name="ELDER_M";
```

returns the string, "ELDER_MELDER_MELDER_M". Since name is a char(24) field in the emp table, without the nested RTRIM, the result would have been 72 characters long and would contain blank spaces between each occurrence of "ELDER_M". If <count> is 0 the result is a zero-length string. If <count> is negative or NULL, or if <string> is NULL, the result is NULL.

2.4.4.10 { fn REPLACE (<s1>, <s2>, <s3>) }

REPLACE replaces all occurrences of <s2> that are found in <s1> with <s3>. For example:

```
select {fn REPLACE(name,"no","NO")} from emp where name="Nobody
Knows";
```

returns the string, "NObody KNOws".

Replacements are made scanning from left to right, and continue scanning just after the replacement string whenever a replacement is made. <s3> is not scanned for occurrences of <s2>. If any argument is NULL, the result is NULL.

2.4.4.11 { fn RIGHT (<str>, <count>) }

RIGHT returns the substring formed by using the last <count> characters from the right in <str>. For example.

```
select { fn RIGHT(name,2) } from emp where name="SMITH"
```

returns the string, "TH". If <count> is greater than the length of <str>, the entire <str> is returned. If <str> or <count> is NULL, then the result is NULL. If <count> is 0, the result is a zero-length string. If <count> is negative, the result is NULL.

2.4.4.12 { fn RTRIM (<string>) }

RTRIM returns <string> with trailing blanks removed.

See {fn REPEAT...} for example. If <string> is NULL, then RTRIM returns NULL.

2.4.4.13 { fn SPACE (<count>) }

SPACE returns <count> number of spaces.

This is equivalent to {fn REPEAT(' ', <count>) }.

2.4.5 Additional Numeric Scalar Functions

2.4.5.1 { fn DEGREES (<radians>) }

DEGREES returns the result of the following formula: `<radians> * (180 / pi)`.

2.4.5.2 { fn EXP (<exp>) }

EXP returns the exponential function of <exp>.

2.4.5.3 { fn LOG (<exp>) }

LOG returns the natural logarithm of <exp>.

2.4.5.4 { fn LOG10 (<exp>) }

LOG10 returns the logarithm base 10 of <exp>.

2.4.5.5 { fn PI () }

PI evaluates to pi expressed to 16 decimal digits (the capacity of a double float) i.e. `3.141592653589793`.

2.4.5.6 { fn POWER (<base>, <exp>) }

Power computes <base> raised to the power <exp>. If <base> is negative then <exp> must be an integer, otherwise the result is NULL. If <base> is 0 and <exp> is negative, the result is NULL.

2.4.5.7 { fn RADIANS (<degrees>) }

RADIANS is defined as <degrees> * (pi / 180) = <degrees> * 0.01745329251994329576

2.4.5.8 { fn RAND ([<int>]) }

RAND returns a random floating point value between 0.0 and 1.0 using <int> as an optional integer seed.

2.4.5.9 { fn ROUND (<number>, <places>) }

ROUND rounds <number> to <places> places to the right of the decimal point. If <places> is negative, then rounding occurs to the left of the decimal point. <Number> is any numeric expression, and <places> is an integer expression.

2.4.5.10 { fn SIGN (<exp>) }

SIGN returns an integer indicating the sign of <exp>. If <exp> is negative, SIGN returns -1. If <exp> is 0, then the result if 0. If <exp> is positive, then the result is 1. If <exp> is NULL, the result is NULL.

2.4.5.11 { fn SQRT (<exp>) }

SQRT returns the square root of <exp>.

2.4.5.12 { fn TRUNCATE (<number>, <places>) }

TRUNCATE will truncate <number> to <places> places to the right of the decimal point. If <places> is negative, then truncation occurs to the left of the decimal point. <Number> is any numeric expression; <places> is an integer expression.

2.4.6 Time and Date Functions

Most arguments may be character string as well as date, time or timestamp. Character string arguments are converted to date, time or timestamp (as appropriate to the function) using the ODBC standard templates, which are `yyyy-mm-dd` for date, `hh:mm:ss` for time, and `yyyy-mm-dd hh:mm:ss.f` for timestamp.

If any argument is NULL or otherwise invalid for the function, the result is NULL.

2.4.6.1 { fn CURDATE () }

CURDATE returns the current date.

2.4.6.2 { fn CURTIME () }

CURTIME returns the current time.

2.4.6.3 { fn DAYOFMONTH (<date>) }

DAYOFMONTH returns the numeric day of the month in <date>. <Date> is a date or timestamp and can be a nested CURDATE().

2.4.6.4 { fn YEAR(<date>) }

YEAR returns the 4 digits of the year in <date>. <Date> is a date or timestamp and can be a nested CURDATE().

2.5 Outer Joins

Outer joins allow records to be retrieved that might not otherwise be retrieved. For example, the following SELECT statement's result set is each employee name along with the corresponding department:

```
select emp.name, dept.name from emp, dept where emp.deptno =
dept.deptno;
```

If there were a department with no employees, that department would not be returned as part of the result set and you therefore may not know it exists. With the OUTERJOIN keyword, all departments would be listed, whether or not there were any employees in that department:

```
select emp.name, dept.name from emp, dept (outerjoin) where emp.deptno
= dept.deptno;
```

In the records where a department has no employees, the employee names would be returned as NULL.

2.5.1 ODBC Outerjoin Syntax

The ODBC syntax for outer joins is:

```
{OJ <table name1> [<correlation name>] LEFT OUTER JOIN <table name2>
[<correlation name2>] ON <search condition>}
```

The search condition may only reference the two tables <table name1> and <table name2> (or their respective correlation names). This construct will be converted internally to standard SQL outerjoin syntax:

```
SELECT ... FROM <table name1> [<correlation name1>], <table name2>
[<correlation name2>] (OUTERJOIN), ... WHERE <search condition>
```

If there are additional non–outerjoined tables, then the conversion is from

```
SELECT ... FROM A, {OJ B LEFT OUTER JOIN C ON <condition1> WHERE
<condition2>
```

to the following standard SQL outerjoin syntax:

```
SELECT ... FROM A, B, C (OUTERJOIN) WHERE <condition1> AND
<condition2>
```

There may be more than one ODBC outerjoin. For example:

```
SELECT ... FROM A, {OJ B LEFT OUTER JOIN C ON <condition1>}, {OJ D
LEFT OUTER JOIN E ON <condition2> WHERE <condition3>
```

becomes:

```
SELECT ... FROM A, B, C (OUTERJOIN), D, E (OUTERJOIN) WHERE
<condition1> AND <condition2> AND <condition3>
```

Nested ODBC outerjoins are not supported.

2.5.2 ODBC Extensions

2.5.2.1 USER literal

The USER keyword can be used in a SELECT list to return the name of the current user as the result or as part of the result of a query. For example, if the current user's username is "williams":

```
SELECT user, name FROM dept WHERE deptno = 1;
```

would result in

```
"williams", "MARKETING"
```

2.5.2.2 ODBC Escape Syntax

{ ESCAPE <character literal> }

This syntax allows definition of an alternate escape character.

2.5.2.3 ODBC Date / Time Literals

The syntax for ODBC date/time literals is as follows:

Date	D 'yyyy-mm-dd'
Time	T 'hh:mm:ss'
Timestamp	{TS 'yyyy-mm-ddhh:mm:ss[.ffffff]'}

2.6 SELECT List CASE Extension

The SQL Select statement supports extensions using the CASE statement to return an alternate set of values. The constructs are:

Case Syntax 1:

```
CASE <test-expression>
  WHEN <when-expression-1> THEN { <result-expression-1> | NULL }
[ WHEN <when-expression-2> THEN { <result-expression-2> | NULL } ]
. . .
[ WHEN <when-expression-n> THEN { <result-expression-n> | NULL } ]
[ ELSE { <final-result-expression> | NULL } ]
END
```

With Syntax 1, the test-expression is evaluated and then compared successively with each when-expression. If one of the comparisons is equality, then the corresponding result-expression is the value of the CASE expression, and no further comparisons are performed.

Note that NULL may not be used as a when-expression, since equality to NULL is never true.

If there is no match, then the final-result-expression in the ELSE clause is the value of the CASE expression. If the ELSE clause is omitted, and no when-expressions match the test-expression, then the value of the CASE expression defaults to NULL.

Case Syntax 2:

```
CASE
  WHEN <condition-1> THEN { <result-expression-1> | NULL }
[ WHEN <condition-2> THEN { <result-expression-2> | NULL}]

. . .
[ WHEN <condition-n> THEN { <result-expression-n> | NULL } } ]
[ ELSE { <final-result-expression> | NULL } ]
END
```

With Syntax 2, the conditions in the WHEN clauses are evaluated successively until one of them evaluates to true. The corresponding result-expression is then the value of the CASE expression and no further tests are performed.

If none of the conditions are true, then the final-result-expression in the ELSE clause is the value of the CASE expression. If the ELSE is omitted and none of the conditions are true, then the CASE expression defaults to NULL.

With both Syntax 1 and Syntax 2, the following restriction applies:

- All result-expressions in a given CASE construct must be the same datatype.

3

Environment Variables

Environment variables are system values that are defined external to the functioning of SQL *Connector* and that can be accessed internally when necessary. For NetWare, these variables are defined using `SQLCADM.NLM` or the SQL *Connector* User Extensions in NetWare Admin.

3.1 Defining Environment Variables

To define an environment variable, use definitions similar to these:

NetWare Console

```
load SQLCADM.NLM
> setenv var_name = var_definition
```

NetWare Admin

Double-Click on a User Object and single click on the SQL User Environment Extension.

Programming

See the *JDBC Programmer's Guide* and the *ODBC Programmer's Guide* for instructions on defining system variables programmatically.

3.2 Environment Variable Names

3.2.1 SS_FETCH_ISOLATION

`SS_FETCH_ISOLATION` is a text string that specifies the isolation level to be used in READ ONLY transactions, and in SELECT, REPORT, and COPYOUT statements outside a transaction. The following settings are used for `SS_FETCH_ISOLATION`:

Setting	Description
NONE	Available only for upward compatibility. Use NONE if you want your SELECT statements to work as in previous versions and you had the LOCK switch set to OFF.
LOCKON	Available only for upward compatibility. Use LOCKON if you want your SELECT statements to work as in previous versions and you had the LOCK switch set to ON.
UNCOMMITTED (default)	Allows dirty reads, nonrepeatable reads, and phantom records.
COMMITTED	Forbids dirty reads; allows nonrepeatable reads and phantom records.

Setting	Description
REPEATABLE	Forbids dirty reads and nonrepeatable reads; allows phantom records.
SERIALIZABLE	Forbids dirty reads, nonrepeatable reads, and phantom records.

3.2.2 SS_ISOLATION

SS_ISOLATION is a text string that specifies the isolation level to be used in update statements (DELETE, INSERT, and UPDATE) outside a transaction, and the default isolation level to be used within a READ WRITE transaction. If you use the ISOLATION LEVEL clause in the BEGIN statement for a transaction, it will override the setting of the ISOLATION switch.

The following settings are available for SS_ISOLATION:

Table 3-1:

Setting	Description
NONE	Available only for upward compatibility. Use NONE if you want your update statements and transactions to work as in previous versions.
COMMITTED	Forbids dirty reads; allows nonrepeatable reads and phantom records.
REPEATABLE (default)	Forbids dirty reads and nonrepeatable reads; allows phantom records.
SERIALIZABLE	Forbids dirty reads, nonrepeatable reads, and phantom records.

3.2.3 SS_MAXBOUND

SS_MAXBOUND is an integer that defines the allowable number of bound variables. The default value is 256. If you intend to allocate more than 256 variables, you must increase the value of this switch before allocating any variables.

3.2.4 SS_MAXFIELDS

SS_MAXFIELDS is an integer that defines the maximum number of fields that are allowable in a SELECT list. The default value is 256. The maximum value allowable is 32767.

3.2.5 SS_MAXTEXT

SS_MAXTEXT is an integer that defines the size of the input query buffer. This value is dynamically increased as needed. The current value can be read from the frontend switch "\WHAT SWITCHES". The initial value is zero.

3.2.6 SS_OJPUSH

SS_OJPUSH is a boolean value. When turned on, SQL *Connector* will push any outerjoin to the database engine to perform whenever possible. When turned off, SQL *Connector*

will perform all outerjoins. The default is on, as better performance is expected when outerjoins are pushed to the native engine.

3.2.7 SS_TRACE

SS_TRACE is a boolean value that enables database tracing. On VMS, output goes to the logical name SMARTTRACE if defined, otherwise to SMARTTRACE.LIS on the default directory. On UNIX and Windows, output goes to the file defined by the SMARTTRACE environment variable. If the SMARTTRACE environment variable is not defined, output goes to the file sDDDHHMM.trc on the current directory, where DDD is the ordinal day of the year, and HHMM is the time of day on a 24-hour clock. SS_TRACE is off by default.

3.2.8 SS_WAIT

SS_WAIT is a boolean value that causes a message to be displayed when a record is locked. By default, the SQL *Connector* transactions will wait for locks (thus possibly causing a delay if a record is currently locked – no messages are displayed). If SS_WAIT is defined to be NOWAIT, then an error message will be received if you attempt to update a locked record.

3.2.9 SMARTTRACE

SMARTTRACE is text string that sets the file specification of the trace file which is created when SS_TRACE is turned on. This can be the full file specification including directory path and filename. If no directory path is given, the file will be created on the current directory.

Index

Symbols

^D input format G–42
^Q input format G–42
^S input format G–42
^T input format G–42
^U input format G–42

A

ABS backend function G–46
action G–10
Aggregates G–43
  AVG G–44
  COUNT G–43
  in subqueries G–45
  MAX G–45
  MIN G–45
  SUM G–44
ALL keyword
  in SELECT statement G–29
AND logical operator G–27
ANY keyword
  in SELECT statement G–29
AVG aggregate G–44

B

Backend Aggregates G–43
Backend functions G–46
  ABS G–46
  CONCAT G–47
  FLOAT4 G–46
  FLOAT8 G–46
  INT2 G–46
  INT4 G–46
  LOWER G–47
  MOD G–46
  NVL G–47
  STRING G–48
  SUBSTRING G–48
  TO_CHAR G–49
  TO_DATE G–49
  UPPER G–49
BETWEEN keyword G–27

C

Case sensitivity G–7
Character strings G–39
Client/Server mode
  COPYIN statement G–9
  COPYOUT statement G–14
Comparison conditions G–27
Compatibility
  non-upward compatible changes G–20
CONCAT backend function G–47
Conversion operators G–46
COPYIN statement G–8
COPYOUT statement G–13, G–15
COUNT aggregate G–43

D

Datatype conversion G–46
Datatypes
  conversion G–21
Date/time input formats G–42
Date/time manipulation
  constants G–40
DCONS G–40
DELETE statement G–18
DESC keyword G–34

E

ESCAPE keyword G–27
EXISTS keyword
  in SELECT statement G–29
Expressions G–39
  arithmetic operators G–39
  date/time constants G–39
  fields G–39
  input date/time formats G–39
  literals G–39

F

FLOAT4 backend function G–46
FLOAT8 backend function G–46
FROM clause G–25
Frontend switches
  MAXBOUND G–57
  MAXFIELDS G–57
  MAXTEXT G–57
  TRACE G–58

G

Grammar components
  aggregate functions G–43
  expressions G–39
GROUP BY clause G–32

GUIDE clause G–35

H

HAVING clause G–33

I

IF G–42
IFNULL backend function G–47
IN keyword
  in SELECT statement G–29
Indicator byte G–11, G–16
Input formats G–42
  ^D G–42
  ^Q G–42
  ^S G–42
  ^T G–42
  ^U UIC G–42
INPUT_DATETIME G–42
INPUT_FORM G–42
INSERT INTO statement G–20
  dates G–21
INT2 backend function G–46
INT4 backend function G–46
ISNULL backend function G–47

J

Joins G–28
  equi-join G–31
  greater-than-join G–31
  null values not matched G–28
  outer joins G–25
  self-join G–31

L

Line feed character
  in Q0delim for COPYIN G–9, G–14
  in S0delim for COPYIN G–9, G–14
Locking
  MINLOCKS option G–25
LOWER backend function G–47

M

MAX aggregate G–45
MAXBOUND frontend switch G–57
MAXFIELDS frontend switch G–57
MAXTEXT frontend switch G–57
MIN aggregate G–45
MINLOCKS G–25
MOD backend function G–46

N

NOT logical operator G–27, G–29
Null values
  aggregates G–45
  arithmetic G–46
  COPYIN statement G–8
  entering G–20, G–37
  exporting G–16
  importing G–11
  INSERT INTO statement G–20
  not equal to null G–28
Numbers G–40
Numeric to string conversion G–46
NVL backend function G–47

O

OLDDATE switch
  with VAXTIME G–41
OR logical operator G–27
ORDER BY clause G–34
OUTERJOIN option G–25

P

Pattern matching G–27

S

s will G–58
Scaled integers
  COPYIN G–11
  COPYOUT G–16
SELECT INTO statement G–36
SELECT statement G–22
  comparison condition G–27
  FROM clause G–25
  GROUP BY clause G–32
  GUIDE clause G–35
  HAVING clause G–33
  join conditions G–28
  ORDER BY clause G–34
  SELECT clause G–23
  SELECT INTO G–36
  subqueries G–28
  WHERE clause G–27
Singleton select G–24
Sorting
  ORDER BY G–34
SQL statements
  COPYIN G–8
  COPYOUT G–13
  DELETE G–18
  INSERT G–20
  SELECT G–22
  UPDATE G–37
SS_ISOLATION logical G–57
STRING backend function G–48
String to numeric conversion G–46
Subqueries G–28

with aggregates G–45
SUBSTRING backend function G–48
SUM aggregate G–44

T

Table access
setting G–25
Tables
maximum number G–25
Template G–42
TO_CHAR backend function G–49
TO_DATE backend function G–49
TRACE frontend switch G–58

U

UPDATE statement G–37
UPPER backend function G–49
USING clause
DELETE statement G–18
UPDATE statement G–37

W

WHERE clause G–27
Wild cards G–27