SQL Reference Guide

November 2004

Version 9.0

This manual describes syntax and semantics of SQL language statements and elements for the Dharma SDK.

November, 2004

Dharma Systems welcomes your comments on this document and the software it describes. Send comments to:

Documentation Comments

Dharma Systems, Inc.

Brookline Business Center.

#55, Route 13

Brookline, NH 03033

Phone: 603-732-4001

Fax: 603-732-4003

Electronic Mail: support@dharma.com

Web Page: http://www.dharma.com

Contents

Introduction

PURPOSE OF THIS GUIDE

This manual details SQL language support provided by the Dharma SDK

AUDIENCE

This manual is intended for application programmers writing database applications using the Dharma SDK environment. To get the most out of this manual, you should be familiar with the concepts of the Dharma SDK environment. If you are new to Dharma SDK, see the *Dharma SDK User's Guide* for supplementary information.

Unless otherwise noted, you can use the SQL syntax described in this manual in the Dharma SDK interactive SQL environment. For more information on interactive SQL see the *Dharma SDK ISQL Manual*.

STRUCTURE

This guide contains the following chapters:

Chapter 1	Describes SQL language elements.
Chapter 2	Provides detailed reference material on each SQL statement, in alphabetic order.
Appendix A	Lists reserved words.
Appendix B	Lists error messages.
Appendix C	Lists the maximum sizes for various attributes of the Dharma SDK Server environment.
Appendix D	Lists the glossary.

SYNTAX DIAGRAM CONVENTIONS

UPPERCASE	Uppercase type denotes reserved words. You must include reserved words in statements, but they can be upper or lower case.
lowercase	Lowercase type denotes either user-supplied elements or names of other syntax diagrams. User-supplied elements include names of tables, host-language variables, expressions and literals. Syntax diagrams can refer to each other by name. If a diagram is named, the name appears in lowercase type above and to the left of the diagram, followed by a double-colon (for example, privilege ::). The name of that diagram appears in lowercase in diagrams that refer to it.
{ }	Braces denote a choice among mandatory elements. They enclose a set of options, separated by vertical bars (|). You must choose at least one of the options.
[ ]	Brackets denote an optional element or a choice among optional elements.
|	Vertical bars separate a set of options.
...	A horizontal ellipsis denotes that the preceding element can optionally be repeated any number of times.
( ) , ;	Parentheses and other punctuation marks are required elements. Enter them as shown in syntax diagrams.

RELATED DOCUMENTATION

Refer to the following guides for more information:

Dharma SDK User Guide	This manual describes the Dharma Software Development Kit (SDK).It describes implementing JDBC, ODBC and .NET access to proprietary data and considerations for creating a release kit to distribute the completed implementation.
Dharma SDK ISQL Reference Manual	This manual provides reference material for the ISQL interactive tool provided in the Dharma SDK environment. It also includes a tutorial describing how to use the ISQL utility.
Dharma SDK ODBC Driver Guide	This manual describes Dharma SDK support for ODBC (Open Database Connectivity) interface and how to configure the Dharma SDK ODBC Driver.
Dharma SDK JDBC Driver Guide	Describes Dharma SDK support for the JDBC interface and how to configure the Dharma SDK JDBC Driver.

Dharma SDK .NET Data Provider Guide	This guide gives an overview of the .NET Data Provider. It describes how to set up and use the .NET Data Provider to access a Dharma SDK database from .NET applications.

Chapter 1

SQL Language Elements

1.1 INTRODUCTION

This chapter describes language elements that are common to many SQL statements. Syntax diagrams in other chapters often refer to these language elements without detailed explanation. The major syntax elements described in the following sections are:

Identifiers (page 1-3) — User-supplied names for elements such as tables, views, cursors, and columns. SQL statements use those names to refer to the elements.

Data types (page 1-4) — Control how SQL stores column values. CREATE TABLE statements specify data types for columns.

Query expressions (page 1-14) — Retrieve values from tables. Query expressions form the basis of other SQL statements and syntax elements

Search conditions (page 1-24) — Specify a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements specify search conditions to restrict the number of rows in the result table.

Expressions (page 1-31) — A symbol or string of symbols used to represent or calculate a single value in an SQL statement. When SQL encounters an expression, it retrieves or calculates the value represented by the expression and uses that value when it executes the statement.

Literals (page 1-36) — A type of SQL expression that specify a constant value. Some SQL constructs allow literals but prohibit other forms of expressions.

Date-time format strings (page 1-40) — Control the output of date and time values. Dharma SDK interprets format strings and replaces them with formatted values.

Functions (page 1-43) — A type of SQL expression that returns a value based on the arguments supplied. Aggregate functions calculate a single value for a collection of rows in a result table. Scalar functions calculate a value based on another single value.

Note:-Information given in this document on Internationalization or Unicode features such as NCHAR and NVARCHAR datatypes and CHARACTER SET clause are supported only in the Dharma/SQL product suite. It is not applicable for Dharma SDK product. Please contact Dharma support for further details on the feature.

1.2 SQL FEATURES: PROFESSIONAL EDITION

The following table lists the SQL features supported by the Professional edition.

Table 1-1: SQL Features Supported by Professional Edition

SQLFeature
SELECT/INSERT/UPDATE/DELETE
INSERT…SELECT
CREATE/DROP TABLE
CREATE/DROP INDEX
CREATE/DROP SYNONYM
GROUP BY clause
AVG, COUNT, MAX, MIN, SUM
20 SQL data types (including LONG, DATE)
Interactive SQL utility
Transaction support
User authentication
Set operators (UNION/INTERSECT/MINUS)
The UNION, INTERSECT, and MINUS set operators manipulate how SQL returns result sets from multiple query expressions (Section 4.4).
Library of over 100 scalar functions (Section 4.9.2)
HAVING clause: The HAVING clause in query expressions (Section 4.4) specifies a search condition for the row groupings specified in the GROUP BY clause.
Table/column privileges: GRANT (Section 5.11) and REVOKE (Section 5.14) statements.
Outer Joins (Section 4.4.2)
CASE expressions (Sections 4.6.4 and 4.9.2.8)
Derived tables: Derived tables are specified through query expressions in the FROM clause of another query expression (Section 4.4).
Subqueries: Subqueries are query expressions used within a search condition or as an expression. Subqueries can occur in basic predicates (Section 4.5.3), quantified predicates (Section 4.5.4), IN predicates (Section 4.5.10), and expressions (Section 4.6)
View support: CREATE (Section 5.5) and DROP VIEW (Section 5.10) statements.
SET SCHEMA statement: (Section 5.16) changes default qualifier for database objects

- ALTER TABLE statement (referred to in the discussion of data types (Section 1.4) and in the discussion of DROP TABLE (Section 2.9))

- CREATE TABLE statement (Section 2.4): TABLESPACE, PCTFREE, or STORAGE_MANAGER clauses
- CREATE INDEX statement (Section 2.2): PCTFREE clause

1.3 SQL IDENTIFIERS

SQL syntax requires users to supply names for elements such as tables, views, cursors, and columns when they define them. SQL statements must use those names to refer to the table, view, or other element. In syntax diagrams, SQL identifiers are shown in lowercase type.

The maximum length for SQL identifiers is 32 characters.

There are two types of SQL identifiers:

- Conventional identifiers
- Delimited identifiers enclosed in double quotation marks

1.3.1 Conventional Identifiers

Unless they are delimited identifiers (see section 1.3.2), SQL identifiers must:

- Begin with an uppercase or lowercase letter
- Contain only letters, digits, or the underscore character (_)
- Not be reserved words

Except for delimited identifiers, SQL does not distinguish between uppercase and lowercase letters in SQL identifiers. It converts all names to lower case, but statements can refer to the names in mixed case. The following examples show some of the characteristics of conventional identifiers:

```
-- Names are case-insensitive:
CREATE TABLE TeSt (CoLuMn1 CHAR);
INSERT INTO TEST (COLUMN1) VALUES('1');
1 record inserted.
SELECT * FROM TEST;
COL
---
1
1 record selected
 TABLE TEST;
COLNAME                         NULL ?       TYPE        LENGTH
-------                         ------       ----        ------
column1
-- Cannot use reserved words:
CREATE TABLE TABLE (COL1 CHAR);
CREATE TABLE TABLE (COL1 CHAR);
```

```
                *
error(-20003): Syntax error
```

1.3.2 Delimited Identifiers

Delimited identifiers are SQL identifiers enclosed in double quotation marks ("). Enclosing a name in double quotation marks preserves the case of the name and allows it to be a reserved word and special characters. (Special characters are any characters other than letters, digits, or the underscore character.) Subsequent references to a delimited identifier must also use enclosing double quotation marks. To include a double-quotation-mark character in a delimited identifier, precede it with another double-quotation mark.

The following SQL example shows some ways to create and refer to delimited identifiers:

```
CREATE TABLE "delimited ids"
      ( """"         CHAR(10),
        "_uscore"    CHAR(10),
        """quote"    CHAR(10),
        " space"     CHAR(10) );
INSERT INTO "delimited ids" (""""") VALUES('text string');
1 record inserted.
SELECT * FROM "delimited ids";
"            _USCORE      "QUOTE        SPACE
-            -------      ------       ------
text strin
1 record selected
CREATE TABLE "TABLE" ("CHAR" CHAR);
```

1.4 DATA TYPES

The SQL statements CREATE TABLE and ALTER TABLE specify data types for each column in the tables they define. This section describes the data types SQL supports for table columns.

There are several categories of SQL data types:

- Character
- Exact numeric
- Approximate numeric
- Date-time
- Bit String

All the data types can store null values. A null value indicates that the value is not known and is distinct from all non-null values.

Syntax

```
data_type ::
      char_data_type
|     exact_numeric_data_type
|     approx_numeric_data_type
|     date_time_data_type
|     bit_string_data_type
```

1.4.1 Character Data Types

See *Character String Literals* on page 1-36 for details on specifying values to be stored in character columns.

Syntax

```
char_data_type ::
     { CHARACTER | CHAR } [(length)] [ CHARACTER SET charset-name ]
|    { CHARACTER VARYING | CHAR VARYING | VARCHAR } [(length)]
          [ CHARACTER SET charset-name ]
|    LVARCHAR | LONG VARCHAR
|    { NATIONAL CHARACTER | NATIONAL CHAR | NCHAR } [(length)]
|    { NATIONAL CHARACTER VARYING | NATIONAL VARCHAR } [(length)]
```

Arguments

{ CHARACTER | CHAR } [(length)] [CHARACTER SET charset-name]

Type CHARACTER (abbreviated as CHAR) corresponds to a null terminated character string with the maximum length specified. The default length is 1. The maximum length is 2000.

The optional CHARACTER SET clause specifies an alternative character set supported by the underlying storage system. *Specifying the Character Set for Character Data Types* on page 1-6 describes general considerations for using this clause. See the documentation for the underlying storage system for details on valid values for charset-name, if any.

{ NATIONAL CHARACTER | NATIONAL CHAR | NCHAR } [(length)]

Type NATIONAL CHARACTER is equivalent to type CHARACTER with a CHARACTER SET clause specifying the character set designated as NATIONAL CHARACTER by the underlying storage system. See *Specifying the Character Set for Character Data Types* on page 1-6.

{ CHARACTER VARYING | CHAR VARYING | VARCHAR } [(length)]
[CHARACTER SET charset-name]

Type CHARACTER VARYING corresponds to a variable-length character string with the maximum length specified.

The optional CHARACTER SET clause specifies an alternative character set supported by the underlying storage system. *Specifying the Character Set for Character*

Data Types on page 1-6 describes general considerations for using this clause. See the documentation for the underlying storage system for details on valid values for char-set-name, if any.

The default length for columns defined as CHARACTER VARYING is 1. The maximum length depends on whether the data type specification includes the CHARACTER SET clause:

- If it does not specify CHARACTER SET, the maximum length is 2000.
- If it does specify CHARACTER SET, the maximum length is 32752.

{ NATIONAL CHARACTER VARYING | NATIONAL VARCHAR } [(length)]
Type NATIONAL CHARACTER VARYING is equivalent to type CHARACTER VARYING with a CHARACTER SET clause specifying the character set designated as NATIONAL CHARACTER by the underlying storage system. See *Specifying the Character Set for Character Data Types* on page 1-6.

LVARCHAR | LONG VARCHAR
Type LONG VARCHAR corresponds to an arbitrarily-long character string with a maximum length limited by the specific storage system.

The arbitrary size and unstructured nature of LONG data types restrict where they can be used.

- LONG columns are allowed in select lists of query expressions and in INSERT statements.
- INSERT statements can store data from columns of any type into a LONG VARCHAR column, but LONG VARCHAR data cannot be stored in any other type.
- CONTAINS predicates are the only predicates that allow LONG columns (and then only if the underlying storage system explicitly supports CONTAINS predicates).
- Conditional expressions, arithmetic expressions, and functions cannot specify LONG columns.
- UPDATE statements cannot specify LONG columns.

1.4.1.1 Specifying the Character Set for Character Data Types

SQL allows column definitions of type CHARACTER and CHARACTER VARYING to specify an alternate character set. If you omit the CHARACTER SET clause in a column definition, the default character set is the standard 7-bit ASCII character set, shown in Table 1-2.

The character set associated with a table column defines which set of characters can be stored in that column, how those characters are represented in the underlying storage system, and how character strings using the character set compare with each other:

- The set of characters allowed in a character set is called the repertoire of the character set. The default ASCII character set has a repertoire of 128 characters, shown in Table 1-2. Other character sets, such as Unicode, specify much larger repertoires and include characters for many languages other than English.

- The storage representation for a character set is called the form of use of the character set. The form of use for the default ASCII character set is a single byte (or octet) containing a number designating a particular ASCII character, also shown in Table 1-2. Other character sets, such as Unicode, use two or more bytes (or a varying number of bytes, depending on the character) for each character.
- The rules used to control how character strings compare with each other is called the collation of a character set. Each character set specifies a collating sequence that defines relative values of each character for comparing, merging and sorting character strings. Character sets may also define additional collations that override the default for a character set. SQL statements specify such collations with the COLLATE clause in character column definitions, basic predicates, the GROUP BY clause of query expressions, and the ORDER BY clause of SELECT statements.

The following table shows the characters in the default ASCII character set and the decimal values that designate each character. (This is the default representation on UNIX; other operating systems may have slight differences in their definitions of the default ASCII character set.) The values also define the collating sequence for the character set. For instance, this collating sequence specifies that a lowercase letter is always a larger value than an uppercase letter.

Table 1-2: Default ASCII Character Set

Val	Char	Val	Char	Val	Char	Val	Char	Val	Char
0	NUL	1	SOH	2	STX	3	ETX	4	EOT
5	ENQ	6	ACK	7	BEL	8	BS	9	HT
10	NL	11	VT	12	NP	13	CR	14	SO
15	SI	16	DLE	17	DC1	18	DC2	19	DC3
20	DC4	21	NAK	22	SYN	23	ETB	24	CAN
25	EM	26	SUB	27	ESC	28	FS	29	GS
30	RS	31	US	32	SP	33	!	34	"
35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,
45	-	46	.	47	/	48-57	0-9	58	:
59	;	60	<	61	=	62	>	63	?
64	@	91	[	92	\	93	]	94	^
95	_	96		`65-90	A-Z	97-122	a-z	123	{
124	|	125	}	126	~	127	DEL		

Dharma SDK supports the ASCII_SET character-set name and a collation sequence named CASE_INSENSITIVE. The ASCII_SET character set is the same as the default and is provided to test and illustrate the CHARACTER SET syntax. The CASE_INSENSITIVE collation sequence overrides the default ASCII collation and

specifies the same comparison values for a lowercase letter as its uppercase counterpart.

The following example uses the ISQL TABLE command to show two tables. bigwigs is defined with the CASE_INSENSITIVE collating sequence for its column, and bigwigs2 is not. Both tables contain the same data. SELECT statements show the difference in collation:

```
ISQL> TABLE bigwigs
COLNAME        NULL?     TYPE      LENGTH    CHARSET NAME   COLLATION
-------        ------    ----      ------    ------------   ---------
name                     CHAR      10
CASE_INSENSITIVE
ISQL> TABLE bigwigs2
COLNAME        NULL?     TYPE      LENGTH    CHARSET NAME   COLLATION
-------        ------    ----      ------    ------------   ---------
name                     CHAR      10
ISQL> select * from bigwigs order by name;
NAME
----
bill
LARRY
mARk
scott
4 records selected
ISQL> select * from bigwigs2 order by name;
NAME
----
LARRY
bill
mARk
scott
```

Support for character sets other than ASCII_SET and collations other than CASE_INSENSITIVE depends on the underlying storage system. When statements refer to a character set or collation name that is not supported by the underlying storage system, SQL generates an error:

```
ISQL> create table badset (c1 char(10) character set bad_set);
create table badset (c1 char(10) character set bad_set);
                                                *
error(-20239): Invalid character set name specified
ISQL> create table badseq (c1 char(10) collate bad_seq);
create table badseq (c1 char(10) collate bad_seq);
                                        *
error(-20240): Invalid collation name specified
```

The NATIONAL CHARACTER reserved words in SQL are shorthand notation for specifying a particular character set supported by the underlying storage system. If

the underlying storage system designates a supported character set as the national character set, column definitions can use the NATIONAL CHARACTER (or NATIONAL CHARACTER VARYING) data type instead of explicitly specifying the character set name in the CHARACTER SET clause of the CHARACTER (or CHARACTER VARYING) data type. If the underlying storage system does not associate another character set with the NATIONAL CHARACTER clause, the default national character set is the ASCII_SET character set.

1.4.2 Exact Numeric Data Types

See *Numeric Literals* on page 1-36 for details on specifying values to be stored in numeric columns.

Syntax

```
exact_numeric_data_type ::
     TINYINT
|    SMALLINT
|    INTEGER
|    BIGINT
|    NUMERIC | NUMBER [ ( precision [ , scale ] ) ]
|    DECIMAL [(precision, scale)]
|    MONEY [(precision)]
```

Arguments

TINYINT

Type TINYINT corresponds to an integer value stored in one byte. The range of TINYINT is -128 to 127.

SMALLINT

Type SMALLINT corresponds to an integer value of length 2 bytes.

The range of SMALLINT is -32768 to +32767.

INTEGER

Type INTEGER corresponds to an integer of length 4 bytes.

The range of values for INTEGER columns is -2 ** 31 to 2 ** 31 -1.

BIGINT

Type BIGINT corresponds to an integer of length 8 bytes. The range of values for BIGINT columns is -2 ** 63 to 2 ** 63 -1.

NUMERIC | NUMBER [(precision [, scale])]

Type NUMERIC corresponds to a number with the given precision (maximum number of digits) and scale (the number of digits to the right of the decimal point). By default, NUMERIC columns have a precision of 32 and scale of 0. If NUMERIC columns omit the scale, the default scale is 0.

The range of values for a NUMERIC type column is -n to +n where n is the largest number that can be represented with the specified precision and scale. If a value exceeds the precision of a NUMERIC column, SQL generates an overflow error. If a value exceeds the scale of a NUMERIC column, SQL rounds the value.

NUMERIC type columns cannot specify a negative scale or specify a scale larger than the precision.

The following example shows what values will fit in a column created with a precision of 3 and scale of 2:

```
insert into t4 values(33.33);
error(-20052): Overflow error
insert into t4 values(33.9);
error(-20052): Overflow error
insert into t4 values(3.3);
1 record inserted.
insert into t4 values(33);
error(-20052): Overflow error
insert into t4 values(3.33);
1 record inserted.
insert into t4 values(3.33333);
1 record inserted.
insert into t4 values(3.3555);
1 record inserted.
select * from t4;
    C1
    --
  3.30
  3.33
  3.33
  3.36
4 records selected
```

DECIMAL [(precision, scale)]

Type DECIMAL is equivalent to type NUMERIC.

MONEY [(precision)]

Type MONEY is equivalent to type NUMERIC with a fixed scale of 2.

1.4.2.1 Approximate Numeric Data Types

See *Numeric Literals* on page 1-36 for details on specifying values to be stored in numeric columns.

Syntax

```
approx_numeric_data_type ::
     REAL
|    DOUBLE PRECISION
|    FLOAT [ (precision) ]
```

Arguments

REAL

Type REAL corresponds to a single precision floating point number equivalent to the C language float type.

DOUBLE PRECISION

Type DOUBLE PRECISION corresponds to a double precision floating point number equivalent to the C language double type.

FLOAT [(precision)]

Type FLOAT corresponds to a double precision floating point number of the given precision. By default, FLOAT columns have a precision of 8.

1.4.3 Date-Time Data Types

See *Date-Time Literals* on page 1-37 for details on specifying values to be stored in date-time columns. See *Date-Time Format Strings* on page 1-41for details on using format strings to specify the output format of date-time columns.

Syntax

```
date_time_data_type ::
    DATE
|   TIME
|   TIMESTAMP
```

Arguments

DATE

Type DATE stores a date value as three parts: year, month, and day. The range for the parts is:

- Year: 1 to 9999
- Month: 1 to 12
- Day: Lower limit is 1; the upper limit depends on the month and the year

TIME

Type TIME stores a time value as four parts: hours, minutes, seconds, and milliseconds. The range for the parts is:

- Hours: 0 to 23
- Minutes: 0 to 59
- Seconds: 0 to 59
- Milliseconds: 0 to 999

TIMESTAMP

Type TIMESTAMP combines the parts of DATE and TIME.

1.4.4 Bit String Data Types

Syntax

```
bit_string_data_type ::
     BIT
|    BINARY [(length)]
|    VARBINARY [(length)]
|    LVARBINARY | LONG VARBINARY
```

Arguments

BIT
Type BIT corresponds to a single bit value of 0 or 1.

SQL statements can assign and compare values in BIT columns to and from columns of types CHAR, VARCHAR, BINARY, VARBINARY, TINYINT, SMALLINT, and INTEGER. However, in assignments from BINARY, VARBINARY, and LONG VARBINARY, the value of the first four bits must be 0001 or 0000.

No arithmetic operations are allowed on BIT columns.

BINARY [(length)]
Type BINARY corresponds to a bit field of the specified length of bytes. The default length is 1 byte. The maximum length is 2000 bytes.

In interactive SQL, INSERT statements must use a special format to store values in BINARY columns. They can specify the binary values as a bit string, hexadecimal string, or character string. INSERT statements must enclose binary values in single-quote marks, preceded by b for a bit string and x for a hexadecimal string:

	Prefix	Suffix	Example (for same 2 byte data)
bit string	b		"b'1010110100010000'
hex string	x		"x'ad10'
char string			"'ad10'

SQL interprets a character string as the character representation of a hexadecimal string.

If the data inserted into a BINARY column is less than the length specified, SQL pads it with zeroes.

BINARY data can be assigned and compared to and from columns of type BIT, CHAR, and VARBINARY types. No arithmetic operations are allowed.

VARBINARY [(length)]

Type VARBINARY corresponds to a variable-length bit field with the maximum length specified. The default length is 1 and the maximum length is 32752. Otherwise, VARBINARY columns have the same characteristics as BINARY.

LVARBINARY | LONG VARBINARY

Type LONG VARBINARY corresponds to an arbitrarily-long bit field with the maximum length defined by the underlying storage system.

The arbitrary size and unstructured nature of LONG data types restrict where they can be used.

- LONG columns are allowed in select lists of query expressions and in INSERT statements.
- INSERT statements can store data from columns of any type into a LONG VARCHAR column, but LONG VARCHAR data cannot be stored in any other type.
- CONTAINS predicates are the only predicates that allow LONG columns (and then only if the underlying storage system explicitly supports CONTAINS predicates).
- Conditional expressions, arithmetic expressions, and functions cannot specify LONG columns.
- UPDATE statements cannot specify LONG columns.

1.5 QUERY EXPRESSIONS

A query expression selects the specified column values from one or more rows contained in one or more tables specified in the FROM clause. The selection of rows is restricted by a search condition in the WHERE clause. The temporary table derived through the clauses of a select statement is called a result table.

Query expressions form the basis of other SQL statements and syntax elements:

- SELECT statements are query expressions with optional ORDER BY and FOR UPDATE clauses.
- CREATE VIEW statements specify their result table as a query expression.
- INSERT statements can specify a query expression to add the rows of the result table to a table.
- UPDATE statements can specify a query expression that returns a single row to modify columns of a row.
- Some search conditions can specify query expressions. Basic predicates can specify query expressions, but the result table can contain only a single value. Quantified and IN predicates can specify query expressions, but the result table can contain only a single column.
- The FROM clause of a query expression can itself specify a query expression, called a derived table.

Syntax

```
query_expression ::
    query_specification
|   query_expression set_operator query_expression
|   ( query_expression )
set_operator ::
     {  UNION [ ALL ]  |  INTERSECT  |  MINUS }
query_specification ::
SELECT [ALL | DISTINCT]
        {
           *
        |  { table_name | alias } . * [, { table_name | alias } . * ] …
        |  expr [ [ AS ] [ ' ] column_title [ ' ] ] [, expr [ [ AS ] [
' ] column_title [ ' ] ] ] ...
        }
FROM table_ref [ { dharma ORDERED } ] [ , table_ref [ { dharma ORDERED
} ] …
[ WHERE search_condition ]
[ GROUP BY [table.]column_name [ COLLATE collation-name ]
                  [, [table.]column_name [ COLLATE collation-name ] ]
...
[ HAVING search_condition ]
table_ref ::
           table_name [ AS ] [ alias [ ( column_alias [ , … ] ) ] ]
```

```
       |  ( query_expression ) [ AS ] alias [ ( column_alias [ , … ] ) ]
        |  [ ( ] joined_table [ ) ]
joined_table ::
           table_ref CROSS JOIN table_ref
        |  table_ref [ INNER | LEFT [ OUTER ] ] JOIN table_ref ON search_condition
```

Arguments

SELECT [ALL | DISTINCT]

DISTINCT specifies that the result table omits duplicate rows. ALL is the default, and specifies that the result table includes all rows.

SELECT * | { table_name | alias } . *

Specifies that the result table includes all columns from all tables named in the FROM clause. For instance, the following examples both specify all the columns in the customers table:

```
SELECT * FROM  customers;
SELECT customers.* FROM  customers;
```

The *tablename.** syntax is useful when the select list refers to columns in multiple tables, and you want to specify all the columns in one of those tables:

```
SELECT CUSTOMERS.CUSTOMER_ID, CUSTOMERS.CUSTOMER_NAME, ORDERS.*
    FROM CUSTOMERS, ORDERS …
```

SELECT expr [[AS] ['] column_title [']]

Specifies a list of expressions, called a select list, whose results will form columns of the result table. Typically, the expression is a column name from a table named in the FROM clause. The expression can also be any supported mathematical expression, scalar function, or aggregate function that returns a value.

The optional ' *column_title* ' argument specifies a new heading for the associated column in the result table. Enclose the new title in single or double quotation marks if it contains spaces or other special characters:

```
SELECT order_value, order_value * .2 AS 'order "markup"' FROM orders;
                      ORDER_VALUE                    ORDER "MARKUP"
                     -----------                    --------------
                       5000000.00                        1000000.00
                        110000.00                          22000.00
                       3300000.00                         660000.00
```

You can qualify column names with the name of the table they belong to:

```
SELECT CUSTOMER.CUSTOMER_ID FROM CUSTOMERS
```

You must qualify a column name if it occurs in more than one table specified in the FROM clause:

```
SELECT CUSTOMERS.CUSTOMER_ID
    FROM CUSTOMERS, ORDERS
```

Qualified column names are always allowed even when they are not required.

FROM table_ref …

The FROM clause specifies one or more table references. Each table reference resolves to one table (either a table stored in the database or a virtual table resulting from processing the table reference) whose rows the query expression uses to create the result table. There are three forms of table references:

- A direct reference to a table, view or synonym
- A derived table specified by a query expression in the FROM clause
- A joined table that combines rows and columns from multiple tables

The usage notes specific to each form of table reference follow.

If there are multiple table references, SQL joins the tables to form an intermediate result table that is used as the basis for evaluating all other clauses in the query expression. That intermediate result table is the Cartesian product of rows in the tables in the FROM clause, formed by concatenating every row of every table with all other rows in all tables.

FROM table_name [AS] [alias [(column_alias [, …])]]

Explicitly names a table. The name listed in the FROM clause can be a table name, a view name, or a synonym.

The alias is a name you use to qualify column names in other parts of the query expression. Aliases are also called correlation names.

If you specify an alias, you must use it, and not the table name, to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

For example, the following query expression joins the table customer with itself. It uses the aliases x and y and returns information on customers in the same city as customer *SMITH*:

```
SELECT y.cust_no, y.name
     FROM customer x, customer y
     WHERE  x.name = 'SMITH'
        AND y.city = x.city ;
```

Similar to table aliases, the *column_alias* provides an alternative name to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in *table_name*. Also, if you specify column aliases in the FROM clause, you must use them-not the column names-in references to the columns.

FROM (query_expression) [AS] alias [(column_alias [, …])]

Specifies a derived table through a query expression. With derived tables, you must specify an alias to identify the derived table.

Derived tables can also specify column aliases. Column aliases provides an alternative name to use in column references elsewhere in the query expression. If you specify column aliases, you must specify them for all the columns in the result table of the

query expression. Also, if you specify column aliases in the FROM clause, you must use them, and not the column names, in references to the columns.

FROM [( ] joined_table [ )]

Combines data from two table references by specifying a join condition. The syntax currently allowed in the FROM clause supports only a subset of possible join conditions:

- CROSS JOIN specifies a Cartesian product of rows in the two tables
- INNER JOIN specifies an inner join using the supplied search condition
- LEFT OUTER JOIN specifies a left outer join using the supplied search condition

You can also specify these and other join conditions in the WHERE clause of a query expression. See *Inner Joins and Outer Joins* on page 1-20 for more detail on both ways of specifying joins.

{ dharma ORDERED }

Directs the SQL engine optimizer to join the tables in the order specified. Use this clause when you want to override the SQL engine's join-order optimization. This is useful for special cases when you know that a particular join order will result in the best performance from the underlying storage system. Since this clause bypasses join-order optimization, carefully test queries that use it to make sure the specified join order is faster than relying on the optimizer.

Note that the braces ({ and }) are part of the required syntax and not syntax conventions.

```
SELECT sc.tbl 'Table', sc.col 'Column',
  sc.coltype 'Data Type', sc.width 'Size'
FROM systpe.syscolumns sc, systpe.systables st
   { dharma ORDERED }
WHERE sc.tbl = st.tbl AND st.tbltype = 'S'
ORDER BY sc.tbl, sc.col;
```

WHERE search_condition

The WHERE clause specifies a *search_condition* that applies conditions to restrict the number of rows in the result table. If the query expression does not specify a WHERE clause, the result table includes all the rows of the specified table reference in the FROM clause.

The *search_condition* is applied to each row of the result table set of the FROM clause. Only rows that satisfy the conditions become part of the result table. If the result of the *search_condition* is NULL for a row, the row is not selected.

Search conditions can specify different conditions for joining two or more tables. See *Inner Joins* on page 1-20 and *Outer Joins* on page 1-22 for more details.

See *Search Conditions* on page 1-24 for details on the different kinds of search conditions.

```
SELECT *
     FROM customer
```

```
WHERE city = 'BURLINGTON' AND state = 'MA' ;
SELECT *
FROM  customer
WHERE  city IN (
     SELECT city
     FROM customer
     WHERE name = 'SMITH') ;
```

GROUP BY column_name ...

Specifies grouping of rows in the result table:

- For the first column specified in the GROUP BY clause, SQL arranges rows of the result table into groups whose rows all have the same values for the specified column.
- If a second GROUP BY column is specified, SQL then groups rows in each main group by values of the second column.
- SQL groups rows for values in additional GROUP BY columns in a similar fashion.

All columns named in the GROUP BY clause must also be in the select list of the query expression. Conversely, columns in the select list must also be in the GROUP BY clause or be part of an aggregate function.

If *column_name* refers to a character column, the column reference can include an optional COLLATE clause. The COLLATE clause specifies a collation sequence supported by the underlying storage system. (See *Specifying the Character Set for Character Data Types* on page 1-6 for notes on character sets and collations. See the documentation for your underlying storage system for details on any supported collations.)

HAVING search_condition

The HAVING clause allows conditions to be set on the groups returned by the GROUP BY clause. If the HAVING clause is used without the GROUP BY clause, the implicit group against which the search condition is evaluated is all the rows returned by the WHERE clause.

A condition of the HAVING clause can compare one aggregate function value with another aggregate function value or a constant.

```
-- select customer number and number of orders for all
-- customers who had more than 10 orders prior to
-- March 31st, 1991.
SELECT  cust_no, count(*)
    FROM  orders
    WHERE order_date < to_date ('3/31/1991')
    GROUP BY cust_no
    HAVING count (*) > 10 ;
```

UNION [ALL]

Appends the result table from one query expression to the result table from another.

The two query expressions must have the same number of columns in their result table, and those columns must have the same or compatible data types.

The final result table contains the rows from the second query expression appended to the rows from the first. By default, the result table does not contain any duplicate rows from the second query expression. Specify UNION ALL to include duplicate rows in the result table.

```
-- Get a merged list of customers and suppliers.
    SELECT name, street, state, zip
    FROM customer
    UNION
    SELECT name, street, state, zip
    FROM supplier ;
-- Get a list of customers and suppliers
-- with duplicate entries for those customers who are
-- also suppliers.
    SELECT name, street, state, zip
    FROM customer
    UNION ALL
    SELECT name, street, state, zip
    FROM supplier ;
```

INTERSECT

Limits rows in the final result table to those that exist in the result tables from both query expressions.

The two query expressions must have the same number of columns in their result table, and those columns must have the same or compatible data types.

```
-- Get a list of customers who are also suppliers.
    SELECT name, street, state, zip
    FROM customer
    INTERSECT
    SELECT name, street, state, zip
    FROM supplier ;
```

MINUS

Limits rows in the final result table to those that exist in the result table from the first query expression minus those that exist in the second. In other words, the MINUS operator returns rows that exist in the result table from the first query expression but that do not exist in the second.

The two query expressions must have the same number of columns in their result table, and those columns must have the same or compatible data types.

```
-- Get a list of suppliers who are not customers.
```

```
SELECT name, street, state, zip
FROM supplier ;
MINUS
SELECT name, street, state, zip
FROM customer;
```

Authorization

The user executing a query expression must have any of the following privileges:

- DBA privilege
- SELECT permission on all the tables/views referred to in the *query_expression*.

SQL Compliance	SQL-92. Extensions: { dharma ORDERED }clause, MINUS set operator
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE TABLE, CREATE VIEW, INSERT, Search Conditions, SELECT, UPDATE

1.5.1 Inner Joins

Description

Inner joins specify how the rows from one table reference are to be joined with the rows of another table reference. Inner joins usually specify a search condition that limits the number of rows from each table reference that become part of the result table generated by the inner join operation.

If an inner join does not specify a search condition, the result table from the join operation is the Cartesian product of rows in the tables, formed by concatenating every row of one table with every row of the other table. Cartesian products (also called cross products or cross joins) are not practically useful, but SQL logically processes all join operations by first forming the Cartesian products of rows from tables participating in the join.

If specified, the search condition is applied to the Cartesian product of rows from the two tables. Only rows that satisfy the search condition become part of the result table generated by the join.

A query expression can specify inner joins in either its FROM clause or in its WHERE clause. For each formulation in the FROM clause, there is an equivalent syntax formulation in the WHERE clause. Currently, not all syntax specified by the SQL-92 standard is allowed in the FROM clause.

Syntax

```
from_clause_inner_join ::
        |  FROM table_ref CROSS JOIN table_ref
        |  FROM table_ref [ INNER ] JOIN table_ref ON
search_condition
```

```
where_clause_inner_join ::
            FROM table_ref, table_ref WHERE search_condition
```

Arguments

FROM table_ref CROSS JOIN table_ref

Explicitly specifies that the join generates the Cartesian product of rows in the two table references. This syntax is equivalent to omitting the WHERE clause and a search condition. The following queries illustrate the results of a simple CROSS JOIN operation and an equivalent formulation that does not use the CROSS JOIN syntax:

```
SELECT * FROM T1;  -- Contents of T1
          C1           C2
          --           --
          10           15
          20           25
2 records selected
SELECT * FROM T2;  -- Contents of T2
          C3 C4
          -- --
          10 BB
          15 DD
2 records selected
SELECT * FROM T1 CROSS JOIN T2; -- Cartesian product
          C1           C2           C3 C4
          --           --           -- --
          10           15           10 BB
          10           15           15 DD
          20           25           10 BB
          20           25           15 DD
4 records selected
SELECT * FROM T1, T2; -- Different formulation, same results
          C1           C2           C3 C4
          --           --           -- --
          10           15           10 BB
          10           15           15 DD
          20           25           10 BB
          20           25           15 DD
4 records selected
```

FROM table_ref [INNER] JOIN table_ref ON search_condition

FROM table_ref, table_ref WHERE search_condition

These two equivalent syntax constructions both specify *search_condition* for restricting rows that will be in the result table generated by the join. In the first format, INNER is optional and has no effect. There is no difference between the WHERE form of inner joins and the JOIN ON form.

Equi-joins

An equi-join specifies that values in one table equal some corresponding column's values in the other:

```
-- For customers with orders, get their name and order info, :
SELECT customer.cust_no, customer.name,
     orders.order_no, orders.order_date
     FROM customers INNER JOIN orders
     ON customer.cust_no = orders.cust_no ;
-- Different formulation, same results:
SELECT customer.cust_no, customer.name,
     orders.order_no, orders.order_date
     FROM customers, orders
     WHERE customer.cust_no = orders.cust_no ;
```

Self joins

A self join, or auto join, joins a table with itself. If a WHERE clause specifies a self join, the FROM clause must use aliases to have two different references to the same table:

```
-- Get all the customers who are from the same city as customer
SMITH:
     SELECT y.cust_no, y.name
     FROM  customer AS x INNER JOIN customer AS y
     ON x.name = 'SMITH' AND y.city = x.city ;
-- Different formulation, same results:
     SELECT y.cust_no, y.name
     FROM  customer x, customer y
     WHERE  x.name = 'SMITH' AND y.city = x.city ;
```

1.5.2 Outer Joins

Description

An outer join between two tables returns more information than a corresponding inner join. An outer join returns a result table that contains all the rows from one of the tables even if there is no row in the other table that satisfies the join condition.

In a left outer join, the information from the table on the left is preserved: the result table contains all rows from the left table even if some rows do not have matching

rows in the right table. Where there are no matching rows in the left table, SQL generates null values.

In a right outer join, the information from the table on the right is preserved: the result table contains all rows from the right table even if some rows do not have matching rows in the left table. Where there are no matching rows in the right table, SQL generates null values.

SQL supports two forms of syntax to support outer joins:

- In the WHERE clause of a query expression, specify the outer join operator (+) after the column name of the table for which rows will not be preserved in the result table. Both sides of an outer-join search condition in a WHERE clause must be simple column references. This syntax is similar to Oracle's SQL syntax, and allows both left and right outer joins.
- For left outer joins only, in the FROM clause, specify the LEFT OUTER JOIN clause between two table names, followed by a search condition. The search condition can contain only the join condition between the specified tables.

Dharma's SQL implementation does not support full (two-sided) outer joins.

Syntax

```
from_clause_inner_join ::
          FROM table_ref LEFT OUTER JOIN table_ref ON
search_condition
where_clause_inner_join ::
          WHERE [table_name.]column (+) = [table_name.]column
        | WHERE [table_name.]column = [table_name.]column (+)
```

Examples

The following example shows a left outer join. It displays all the customers with their orders. Even if there is not a corresponding row in the orders table for each row in the customer table, NULL values are displayed for the orders.*order_no* and *orders.order_date* columns.

```
SELECT customer.cust_no, customer.name, orders.order_no,
            orders.order_date
     FROM customers, orders
     WHERE customer.cust_no = orders.cust_no (+) ;
The following series of examples illustrates the outer join
syntax:
SELECT * FROM T1; -- Contents of T1
C1   C2
--   --
10   15
20   25
2 records selected
SELECT * FROM T2; -- Contents of T2
```

```
C3   C4
--   --
10   BB
15   DD
2 records selected
-- Left outer join
SELECT * FROM T1 LEFT OUTER JOIN T2 ON T1.C1 = T2.C3;
C1   C2   C3   C4
--   --   --   --
10   15   10   BB
20   25
2 records selected
 -- Left outer join: different formulation, same results
 SELECT * FROM T1, T2 WHERE T1.C1 = T2.C3 (+);
C1   C2   C3   C4
--   --   --   --
10   15   10   BB
20   25
2 records selected
 -- Right outer join
 SELECT * FROM T1, T2 WHERE T1.C1 (+) =  T2.C3;
C1   C2   C3   C4
--   --   --   --
10   15   10   BB
          15   DD
2 records selected
```

1.6 SEARCH CONDITIONS

Description

A search condition specifies a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or UPDATE statement.

Search conditions contain one or more predicates. The predicates that can be part of a search condition are described in the following subsections.

Syntax

```
search_condition ::
        [NOT] predicate
        [ { AND | OR } { predicate | ( search_condition ) } ]
predicate ::
```

```
          basic_predicate
|         quantified_predicate
|         between_predicate
|         null_predicate
|         like_predicate
|         contains_predicate
|         exists_predicate
|         in_predicate
|         outer_join_predicate
```

1.6.1 Logical Operators: OR, AND, NOT

Logical operators combine multiple search conditions. SQL evaluates multiple search conditions in this order:

1. Search conditions enclosed in parentheses. If there are nested search conditions in parentheses, SQL evaluates the innermost search condition first.
2. Search conditions preceded by NOT
3. Search conditions combined by AND
4. Search conditions combined by OR

Examples

```
SELECT *
    FROM customer
    WHERE name = 'LEVIEN' OR name = 'SMITH' ;
SELECT *
    FROM customer
    WHERE city = 'PRINCETON' AND state = 'NJ' ;
SELECT *
    FROM customer
    WHERE  NOT (name = 'LEVIEN' OR name = 'SMITH') ;
```

1.6.2 Relational Operators

Relational operators specify how SQL compares expressions in basic and quantified predicates.

Syntax

```
relop ::
       =
     | <> | != | ^=
     | <
     | <=
     | >
```

```
    | >=
```

Relational Operator	Predicate is:
=	True if the two expressions are equal.
<> | != | ^=	True if the two expressions are not equal. The operators != and ^= are equivalent to <>.
<	True if the first expression is less than the second expression..
<=	True if the first expression is less than or equal to the second expression.
>	True if the first expression is greater than the second expression.
>=	True if the first expression is greater than or equal to the second expression.

See *Basic Predicate* on page 1-26 and *Quantified Predicate* on page 1-26 for more information.

1.6.3 Basic Predicate

Description

A basic predicate compares two values using a relational operator (see *Relational Operators* on page 1-25). If a basic predicate specifies a query expression, then the query expression must return a single value. Basic predicates often specify an inner join. See "Inner Joins" for more detail.

If the value of any expression is null or the *query_expression* does not return any value, then the result of the predicate is set to false.

Basic predicates that compare two character expressions can include an optional COLLATE clause. The COLLATE clause specifies a collation sequence supported by the underlying storage system. (See *Specifying the Character Set for Character Data Types* on page 1-6 for notes on character sets and collations. See the documentation for your underlying storage system for details on any supported collations.)

Syntax

```
basic_predicate ::
    expr relop { expr | (query_expression) } [ COLLATE
collation_name ]
```

1.6.4 Quantified Predicate

Description

The quantified predicate compares a value with a collection of values using a relational operator (see Relational Operators on page 1-25). A quantified predicate has the same form as a basic predicate with the *query_expression* being preceded by ALL,

ANY or SOME keyword. The result table returned by *query_expression* can contain only a single column.

When ALL is specified the predicate evaluates to true if the *query_expression* returns no values or the specified relationship is true for all the values returned.

When SOME or ANY is specified the predicate evaluates to true if the specified relationship is true for at least one value returned by the *query_expression*. There is no difference between the SOME and ANY keywords. The predicate evaluates to false if the *query_expression* returns no values or the specified relationship is false for all the values returned.

Syntax

```
quantified_predicate ::
    expr relop { ALL | ANY | SOME } (query_expression)
```

Example

```
10 <  ANY ( SELECT COUNT(*)
                FROM order_tbl
                GROUP BY custid
              )
```

1.6.5 BETWEEN Predicate

Description

The BETWEEN predicate can be used to determine if a value is within a specified value range or not. The first expression specifies the lower bound of the range and the second expression specifies the upper bound of the range.

The predicate evaluates to true if the value is greater than or equal to the lower bound of the range, or less than or equal to the upper bound of the range.

Syntax

```
between_predicate ::
    expr [ NOT ] BETWEEN expr AND expr
```

Example

```
salary BETWEEN 2000.00 AND 10000.00
```

1.6.6 NULL Predicate

Description

The NULL predicate can be used for testing null values of database table columns.

Syntax

```
null_predicate ::
    column_name IS [ NOT ] NULL
```

Example

```
contact_name IS NOT NULL
```

1.6.7 CONTAINS Predicate

Description

The SQL CONTAINS predicate is an extension to the SQL standard that allows storage systems to provide search capabilities on character and binary data. See the documentation for the underlying storage system for details on support, if any, for the CONTAINS predicate.

Syntax

```
column_name [ NOT ] CONTAINS 'string'
```

Notes

- *column_name* must be one of the following data types: CHARACTER, VARCHAR, LONG VARCHAR, BINARY, VARBINARY, or LONG VARBINARY.
- There must be an index defined for *column_name*, and the CREATE INDEX statement for the column must include the TYPE clause, and specify an index type that indicates to the underlying storage system that this index supports CONTAINS predicates. See the documentation for your storage system for the correct TYPE clause for indexes that support CONTAINS predicates.
- The format of the quoted string argument and the semantics of the CONTAINS predicate are defined by the underlying storage system.

1.6.8 LIKE Predicate

Description

The LIKE predicate searches for strings that have a certain pattern. The pattern is specified after the LIKE keyword in a string constant. The pattern can be specified by a string in which the underscore (_) and percent sign (%) characters have special semantics.

The ESCAPE clause can be used to disable the special semantics given to characters ' _ ' and ' % '. The escape character specified must precede the special characters in order to disable their special semantics.

Syntax

```
like_predicate ::
    column_name [ NOT ] LIKE string_constant
    [ ESCAPE escape-character ]
```

Notes

- The column name specified in the LIKE predicate must refer to a character string column.

- A percent sign in the pattern matches zero or more characters of the column string.
- A underscore sign in the pattern matches any single character of the column string.

Examples

```
cust_name LIKE '%Computer%'
cust_name LIKE '___'
item_name LIKE '%\_%' ESCAPE '\'
```

In the first example, for all strings with the substring Computer, the predicate will evaluate to true. In the second example, for all strings which are exactly three characters long, the predicate will evaluate to true. In the third example, the backslash character '\' has been specified as the escape character, which means that the special interpretation given to the character '_' is disabled. The pattern will evaluate to TRUE if the column *item_name* has embedded underscore characters.

1.6.9 EXISTS Predicate

Description

The EXISTS predicate can be used to check for the existence of specific rows. The *query_expression* returns rows rather than values. The predicate evaluates to true if the number of rows returned by the *query_expression* is non zero.

Syntax

```
exists_predicate ::
    EXISTS ( query_expression )
```

Example

```
EXISTS (SELECT * FROM order_tbl
                WHERE order_tbl.custid = :custid)
```

In this example, the predicate will evaluate to true if the specified customer has any orders.

1.6.10 IN Predicate

Description

The IN predicate can be used to compare a value with a set of values. If an IN predicate specifies a query expression, then the result table it returns can contain only a single column.

Syntax

```
in_predicate ::
    expr [ NOT ] IN { ( query_expression ) |
                              ( constant , constant [ , ... ] ) }
```

Example

```
address.state IN ('MA', 'NH')
```

1.6.11 Outer Join Predicate

Description

An outer join predicate specifies two tables and returns a result table that contains all of the rows from one of the tables, even if there is no matching row in the other table. See *Outer Joins* on page 1-22 for more information.

Syntax

```
outer_join_predicate ::
    [ table_name. ] column = [ table_name. ] column (+)
    |  [table_name. ] column (+) = [ table_name. ] column
```

1.7 EXPRESSIONS

Description

An expression is a symbol or string of symbols used to represent or calculate a single value in an SQL statement. When you specify an expression in a statement, SQL retrieves or calculates the value represented by the expression and uses that value when it executes the statement.

Expressions are also called scalar expressions or value expressions.

Syntax

```
expr ::
     [ { table_name | alias } . ] column-name
|    character-literal
|    numeric-literal
|    date-time-literal
|    aggregate-function
|    scalar-function
|    concatenated-char-expr
|    numeric-arith-expr
|    date-arith-expr
|    conditional-expr
|    ( expr )
```

Arguments

[{ table_name | alias } .] column-name
A column in a table.

You can qualify column names with the name of the table they belong to:

```
SELECT CUSTOMER.CUSTOMER_ID FROM CUSTOMERS
```

You must qualify a column name if it occurs in more than one table specified in the FROM clause:

```
SELECT CUSTOMER.CUSTOMER_ID
    FROM CUSTOMERS, ORDERS
```

Qualified column names are always allowed even when they are not required.

You can also qualify column names with an alias. Aliases are also called correlation names.

The FROM clause of a query expression can specify an optional alias after the table name (see *Query Expressions* on page 1-14 for details). If you specify an alias, you must use it-not the table name-to qualify column names that refer to the table. Query expressions that join a table with itself must use aliases to distinguish between references to column names.

The following example shows a query expression that joins the table customer with itself. It uses the aliases x and y and returns information on customers in the same city as customer SMITH:

```
SELECT y.cust_no, y.name
     FROM  customer x, customer y
     WHERE  x.name = 'SMITH'
        AND y.city = x.city ;
```

character-literal | numeric-literal | date-time-literal

Literals that specify a constant value. See *Literals* on page 1-36 for details.

aggregate-function | scalar function

An SQL function. See *Functions* on page 1-43 for details.

concatenated-char-expr

An expression that concatenates multiple character expressions into a single character string. See *Concatenated Character Expressions* on page 1-32 for details.

numeric-arith-expr

An expression that computes a value from numeric values. See *Numeric Arithmetic Expressions* on page 1-33 for details.

date-arith-expr

An expression that computes a value from date-time values. See *Date Arithmetic Expressions* on page 1-34 for details.

conditional-expr

An expression that evaluates a search condition or expression and returns one of multiple possible results depending on that evaluation. See *Conditional Expressions* on page 1-35 for details.

(expr)

An expression enclosed in parentheses. SQL evaluates expressions in parentheses first.

1.7.1 Concatenated Character Expressions

Description

The || concatenation operator (two vertical bars) concatenates the two character expressions it separates.

The concatenation operator is similar to the CONCAT scalar function (see the CONCAT function on page 1-56). However, the concatenation operator allows easy concatenation of more than two character expressions, while the CONCAT scalar function requires nesting.

Syntax

```
concatenated-char-expr ::
{ character-literal | character-expr } || { character-literal |
character-expr }
```

```
[ { character-literal | character-expr } || { character-literal
| character-expr } ] [ … ]
```

Arguments

character-literal

A character literal. See *Character String Literals* on page 1-36 for details.

character-expr

Any expression that evaluates to a character string (see *Data Types* on page 1-4 for details of character data types), including column names and scalar functions that return a character string.

Examples

```
ISQL> SELECT 'Today''s date is ' || TO_CHAR(SYSDATE) FROM
SYSCALCTABLE;
TODAY'S DATE IS 08/17/1998
-------------------------
Today's date is 08/17/1998
1 record selected
```

1.7.2 Numeric Arithmetic Expressions

Description

Numeric arithmetic expressions compute a value using addition, subtraction, multiplication, and division operations on numeric literals and expressions that evaluate to any numeric data type.

Syntax

```
numeric-arith-expr ::
[ + | - ] { numeric-literal | numeric-expr } [ { + | - | * | / }
numeric-arith-expr ]
```

Arguments

[+ | -]

Unary plus or minus operator

numeric-literal

A numeric literal. See *Numeric Literals* on page 1-36 for details.

numeric-expr

Any expression that evaluates to a numeric data type (see *Data Types* on page 1-4 for details of numeric data types), including:

- Column names
- Subqueries that return a single value
- Aggregate functions
- CAST or CONVERT operations to numeric data types

- Other scalar functions that return a numeric data type

{ + | - | * | \ }
Addition, subtraction, multiplication, or subtraction operator. SQL evaluates numeric arithmetic expressions in the following order:

- Unary plus or minus
- Expressions in parentheses
- Multiplication and division, from left to right
- Addition and subtraction, from left to right

1.7.3 Date Arithmetic Expressions

Description

Date arithmetic expressions compute the difference between date-time expressions in terms of days or milliseconds. SQL supports these forms of date arithmetic:

- Addition and subtraction of integers to and from date-time expressions
- Subtraction of a date-time expression from another

Syntax

```
date_arith_expr ::
     date_time_expr { + | - } int_expr
|    date_time_expr - date_time_expr
```

Arguments

date_time_expr
An expression that returns a value of type DATE or TIME or TIMESTAMP. A single date-time expression can not mix data types. All elements of the expression must be the same data type.

Date-time expressions can contain date-time literals, but they must be converted to DATE or TIME using the CAST, CONVERT, or TO_DATE functions (see the following examples, and the CAST function on page 1-53 and the CONVERT function (extension) on page 1-57).

int_expr
An expression that returns an integer value. SQL interprets the integer differently depending on the data type of the date-time expression:

- For DATE expressions, integers represent days
- For TIME expressions, integers represent milliseconds
- For TIMESTAMP expressions, integers represent milliseconds

Examples

The following example manipulates DATE values using date arithmetic. SQL interprets integers as days and returns date differences in units of days:

```
SELECT C1, C2, C1-C2 FROM DTEST
c1           c2           c1-c2
1956-05-07   1952-09-29   1316
select sysdate,
   sysdate - 3 ,
   sysdate - cast ('9/29/52' as date)
from dtest;
sysdate      sysdate-3     sysdate-convert(date,9/29/52)
1995-03-24   1995-03-21    15516
```

The following example manipulates TIME values using date arithmetic. SQL interprets integers as milliseconds and returns time differences in milliseconds:

```
select systime,
       systime - 3000,
       systime - cast ('15:28:01' as time)
from dtest;
systime      systime-3000     systime-convert(time,15:28:01)
15:28:09     15:28:06         8000
```

1.7.4 Conditional Expressions

Conditional expressions are a subset of scalar functions that generate different results depending on the value of their arguments. They provide some of the flexibility of traditional programming constructs to allow expressions to return alternate results depending on the value of their arguments.

The following scalar functions provide support for conditional expressions. See the discussion for each function in *Scalar Functions* on page 1-45 for details.

CASE

CASE is the most general conditional expression. It specifies a series of search conditions and associated expressions. SQL returns the value specified by the first expression whose associated search condition evaluates as true. If none of the expressions evaluate as true, the CASE expression returns a null value (or the value of some other default expression if the CASE expression includes the ELSE clause).

All the other conditional expressions can also be expressed as CASE expressions.

DECODE

DECODE provides a subset of the functionality of CASE that is compatible with Oracle SQL syntax. DECODE is not SQL-92 compatible.

NULLIF

NULLIF substitutes a null value for an expression if it is equal to a second expression.

COALESCE

COALESCE specifies a series of expressions. SQL returns the first expression whose value is not null. If all the expressions evaluate as null, COALESCE returns a null value.

IFNULL
IFNULL substitutes a specified value if an expression evaluates as null. If the expression is not null, IFNULL returns the value of the expression.

1.8 LITERALS

Literals are a type of expression that specify a constant value (they are also called constants). You can specify literals wherever SQL syntax allows expressions. Some SQL constructs allow literals but prohibit other forms of expressions.

There are three types of literals:

- Numeric
- Character string
- Date-time

The following sections discuss each type of literal.

1.8.1 Numeric Literals

A numeric literal is a string of digits that SQL interprets as a decimal number. SQL allows the string to be in a variety of formats, including scientific notation.

Syntax

```
[+|-]{[0-9][0-9]...}[.[0-9][0-9]...][[E|e][+|-][0-9]{[0-9]}]
```

Examples

The following are all valid numeric strings:

123

123.456

-123.456

12.34E-04

1.8.2 Character String Literals

A character string literal is a string of characters enclosed in single quotation marks (').

To include a single quotation mark in a character-string literal, precede it with an additional single quotation mark. The following SQL examples show embedding quotation marks in character-string literals:

```
insert into quote values('unquoted literal');
insert into quote values('''single-quoted literal''');
```

```
insert into quote values('"double-quoted literal"');
insert into quote values('O''Hare');
select * from quote;
c1
unquoted literal
'single-quoted literal'
"double-quoted literal"
O'Hare
```

To insert a character-string literal that spans multiple lines, enclose each line in single quotation marks. The following SQL examples shows this syntax, as well as embedding quotation marks in one of the lines:

```
insert into quote2 values ('Here''s a very long character
string '
    'literal that will not fit on a single line.');
1 record inserted.
select * from quote2;
C1
--
```

Here's a very long character string literal that will not fit on a single line.

1.8.3 Date-Time Literals

SQL supports special formats for literals to be used in conjunction with date-time data types. Basic predicates and the VALUES clause of INSERT statements can specify date literals directly for comparison and insertion into tables. In other cases, you need to convert date literals to the appropriate date-time data type with the CAST, CONVERT, or TO_DATE scalar functions.

Enclose date-time literals in single quotation marks.

1.8.3.1 Date Literals

Date literals specify a day, month, and year. By default, SQL supports any of the following formats, enclosed in single quotation marks ('). Check with your administrator to see if the set of supported formats has been changed by setting the TPE_DFLT_DATE runtime variable.

Syntax

```
date-literal ::
     {d 'yyyy-mm-dd'}
|    mm-dd-yyyy
|    mm/dd/yyyy
|    yyyy-mm-dd
|    yyyy/mm/dd
|    dd-mon-yyyy
```

```
|     dd/mon/yyyy
```

Arguments

{d 'yyyy-mm-dd'}
A date literal enclosed in an escape clause compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase d. End the literal with a close brace. For example:

```
INSERT INTO DTEST VALUES ({d '1994-05-07'})
```

If you use the ODBC escape clause, you must specify the date using the format yyyy-mm-dd.

dd
The day of month as a 1- or 2-digit number (in the range 01-31).

mm
The month value as a 1- or 2-digit number (in the range 01-12).

mon
The first 3 characters of the name of the month (in the range 'JAN' to 'DEC').

yyyy
The year as 4-digit number. By default, SQL generates an Invalid date string error if the year is specified as anything but 4 digits. Check with your administrator to see if this default behavior has been changed by setting the DH_Y2K_CUTOFF runtime variable.

Examples

The following SQL examples show some of the supported formats for date literals:

```
CREATE TABLE T2 (C1 DATE, C2 TIME);
INSERT INTO T2 (C1) VALUES('5/7/56');
INSERT INTO T2 (C1) VALUES('7/MAY/1956');
INSERT INTO T2 (C1) VALUES('1956/05/07');
INSERT INTO T2 (C1) VALUES({d '1956-05-07'});
INSERT INTO T2 (C1) VALUES('29-sEP-1952');
SELECT C1 FROM T2;
c1
1956-05-07
1956-05-07
1956-05-07
1956-05-07
1952-09-29
```

1.8.3.2 Time Literals

Time literals specify an hour, minute, second, and millisecond, using the following format, enclosed in single quotation marks ('):

Syntax

```
time-literal ::
     {t 'hh:mi:ss'}
|    hh:mi:ss[:mls]
```

Arguments

{t 'hh:mi:ss'}
A time literal enclosed in an escape clause compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase t. End the literal with a close brace. For example:

```
INSERT INTO TTEST VALUES ({t '23:22:12'})
```

If you use the ODBC escape clause, you must specify the time using the format hh:mi:ss.

hh
The hour value as a 1- or 2-digit number (in the range 00 to 23).

mi
The minute value as a 1- or 2-digit number (in the range 00 to 59).

ss
The seconds value as a 1- or 2-digit number (in the range 00 to 59).

mls
The milliseconds value as a 1- to 3-digit number (in the range 000 to 999).

Examples

The following SQL examples show some of the formats SQL will and will not accept for time literals:

```
INSERT INTO T2 (C2) VALUES('3');
error(-20234): Invalid time string
INSERT INTO T2 (C2) VALUES('8:30');
error(-20234): Invalid time string
INSERT INTO T2 (C2) VALUES('8:30:1');
INSERT INTO T2 (C2) VALUES('8:30:');
error(-20234): Invalid time string
INSERT INTO T2 (C2) VALUES('8:30:00');
INSERT INTO T2 (C2) VALUES('8:30:1:1');
INSERT INTO T2 (C2) VALUES({t'8:30:1:1'});
SELECT C2 FROM T2;
```

```
c2
08:30:01
08:30:00
08:30:01
08:30:01
```

1.8.3.3 Timestamp Literals

Timestamp literals specify a date and a time separated by a space, enclosed in single quotation marks ('):

Syntax

```
     {ts 'yyyy-mm-dd hh:mi:ss'}
|    ' date-literal time-literal '
```

Arguments

{ts 'yyyy-mm-dd hh:mi:ss'}
A timestamp literal enclosed in an escape clause compatible with ODBC. Precede the literal string with an open brace ({) and a lowercase ts. End the literal with a close brace. For example:

```
INSERT INTO DTEST
VALUES ({ts '1956-05-07 10:41:37'})
```

If you use the ODBC escape clause, you must specify the timestamp using the format yyyy-mm-dd hh:mi:ss.

date-literal
A date literal.

time-literal
A time literal.

Example

```
SELECT * FROM DTEST WHERE C1 = {ts '1956-05-07 10:41:37'}
```

1.9 DATE-TIME FORMAT STRINGS

The TO_CHAR scalar function supports a variety of format strings to control the output of date and time values. The format strings consist of keywords that SQL interprets and replaces with formatted values.

The format strings are case sensitive. For instance, SQL replaces 'DAY' with all uppercase letters, but follows the case of 'Day'.

Supply the format strings, enclosed in single quotation marks, as the second argument to the TO_CHAR function. For example:

```
SELECT C1 FROM T2;
C1
--
```

```
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
         TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
---------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

For details of the TO_CHAR function, see TO_CHAR on page 1-95.

1.9.1 Date Format Strings

A date format string can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals.

CC	The century as a 2-digit number.
YYYY	The year as a 4-digit number.
YYY	The last 3 digits of the year.
YY	The last 2 digits of the year.
Y	The last digit of the year.
Y,YYY	The year as a 4-digit number with a comma after the first digit.
Q	The quarter of the year as 1-digit number (with values 1, 2, 3, or 4).
MM	The month value as 2-digit number (in the range 01-12).
MONTH	The name of the month as a string of 9 characters ('JANUARY' to 'DECEMBER ').
MON	The first 3 characters of the name of the month (in the range 'JAN' to 'DEC').
WW	The week of year as a 2-digit number (in the range 01-52).
W	The week of month as a 1-digit number (in the range 1-5).
DDD	The day of year as a 3-digit number (in the range 001-365).
DD	The day of month as a 2-digit number (in the range 01-31).
D	The day of week as a 1-digit number (in the range 1-7, 1 for Sunday and 7 for Saturday).
DAY	The day of week as a 9 character string (in the range 'SUNDAY' to 'SATURDAY '.
DY	The day of week as a 3 character string (in the range 'SUN' to 'SAT').
J	The Julian day (number of days since DEC 31, 1899) as an 8 digit number.
TH	When added to a format keyword that results in a number, this format keyword ('TH') is replaced by the string 'ST', 'ND', 'RD' or 'TH' depending on the last digit of the number.

Example:

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
         TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
---------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

1.9.2 Time Format Strings

A time format string can contain any of the following format keywords along with other characters. The format keywords in the format string are replaced by corresponding values to get the result. The other characters are displayed as literals.

AM PM	The string 'AM' or 'PM' depending on whether time corresponds to forenoon or afternoon.
A.M. P.M.	The string 'A.M.' or 'P.M.' depending on whether time corresponds to forenoon or afternoon.
HH12	The hour value as a 2-digit number (in the range 00 to 11).
HH HH24	The hour value as a 2-digit number (in the range 00 to 23).
MI	The minute value as a 2-digit number (in the range 00 to 59).
SS	The seconds value as a 2-digit number (in the range 00 to 59).
SSSSS	The seconds from midnight as a 5-digit number (in the range 00000 to 86399).
MLS	The milliseconds value as a 3-digit number (in the range 000 to 999).

Example:

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
```

```
          TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
---------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

1.10 FUNCTIONS

Functions are a type of SQL expression that return a value based on the argument they are supplied. SQL supports two types of functions:

- Aggregate functions calculate a single value for a collection of rows in a result table (if the function is in a statement with a GROUP BY clause, it returns a value for each group in the result table). Aggregate functions are also called set or statistical functions. Aggregate functions cannot be nested.
- Scalar functions calculate a value based on another single value. Scalar functions are also called value functions. Scalar functions can be nested.

1.10.1 Aggregate Functions

1.10.1.1 AVG

Syntax

```
AVG ( { [ALL] expression } | { DISTINCT column_ref } )
```

Description

The aggregate function AVG computes the average of a collection of values. The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the average.

- Null values are eliminated before the average value is computed. If all the values are null, the result is null.
- The argument to the function must be of type SMALLINT, INTEGER, NUMERIC, REAL or FLOAT.
- The result is of type NUMERIC.

Example

```
SELECT AVG (salary)
    FROM employee
    WHERE deptno = 20 ;
```

1.10.1.2 COUNT

Syntax

```
COUNT ( { [ALL] expression } | { DISTINCT column_ref } | * )
```

Description

The aggregate function COUNT computes either the number of rows in a group of rows or the number of non-null values in a group of values.

- The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the count.
- If the argument to COUNT function is '*', then the function computes the count of the number of rows in group.
- If the argument to COUNT function is not '*', then null values are eliminated before the number of rows is computed.
- The argument *column_ref* or expression can be of any type.
- The result of the function is of INTEGER type. The result is never null.

Example

```
SELECT COUNT (*)
    FROM orders
    WHERE order_date = SYSDATE ;
```

1.10.1.3 MAX

Syntax

```
MAX ( { [ALL] expression } | { DISTINCT column_ref } )
```

Description

The aggregate function MAX returns the maximum value in a group of values.

- The specification of DISTINCT has no effect on the result.
- The argument *column_ref* or expression can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is null if the result set is empty or contains only null values.

Example

```
SELECT order_date, product, MAX (qty)
    FROM orders
    GROUP BY order_date, product ;
```

1.10.1.4 MIN

Syntax

```
MIN ( { [ALL] expression } | { DISTINCT column_ref } )
```

Description

The aggregate function MIN returns the minimum value in a group of values.

- The specification of DISTINCT has no effect on the result.
- The argument *column_ref* or expression can be of any type.
- The result of the function is of the same data type as that of the argument.
- The result is null if the result set is empty or contains only null values.

Example

```
SELECT MIN (salary)
    FROM employee
    WHERE deptno = 20 ;
```

1.10.1.5 SUM

Syntax

```
SUM ( { [ALL] expression } | { DISTINCT column_ref } )
```

Description

The aggregate function SUM returns the sum of the values in a group. The keyword DISTINCT specifies that the duplicate values are to be eliminated before computing the sum.

- The argument *column_ref* or expression can be of any type.
- The result of the function is of the same data type as that of the argument except that the result is of type INTEGER when the argument is of type SMALLINT or TINYINT.
- The result can have a null value.

Example

```
SELECT SUM (amount)
    FROM orders
    WHERE order_date = SYSDATE ;
```

1.10.2 Scalar Functions

1.10.2.1 ABS function (ODBC compatible)

Syntax

```
ABS ( expression )
```

Description

The scalar function ABS computes the absolute value of expression.

Example

```
SELECT  ABS (MONTHS_BETWEEN (SYSDATE, order_date))
    FROM  orders
```

```
WHERE   ABS (MONTHS_BETWEEN (SYSDATE, order_date)) > 3 ;
```

Notes

- The argument to the function must be of type TINYINT, SMALLINT, INTEGER, NUMERIC, REAL or FLOAT.
- The result is of type NUMERIC.
- If the argument expression evaluates to null, the result is null.

1.10.2.2 ACOS function (ODBC compatible)

Syntax

```
ACOS ( expression )
```

Description

The scalar function ACOS returns the arccosine of expression.

Example

```
select acos (.5) 'Arccosine in radians' from syscalctable;
ARCCOSINE IN RAD
----------------
1.047197551196598
1 record selected
select acos (.5) * (180/ pi()) 'Arccosine in degrees' from
syscalctable;
ARCCOSINE IN DEG
----------------
59.999999999999993
1 record selected
```

Notes

ACOS takes the ratio (expression) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

- expression must be in the range -1 to 1.
- expression must evaluate to an approximate numeric data type.

1.10.2.3 ADD_MONTHS function (extension)

Syntax

```
ADD_MONTHS ( date_expression, integer_expression )
```

Description

The scalar function ADD_MONTHS adds to the date value specified by the *date_expression*, the given number of months specified by *integer_expression* and returns the resultant date value.

Example

```
SELECT *
FROM   customer
WHERE  ADD_MONTHS (start_date, 6) > SYSDATE ;
```

Notes

- The first argument must be of DATE type.
- The second argument to the function must be of numeric type.
- The result is of type DATE.
- If any of the arguments evaluate to null, the result is null.

1.10.2.4 ASCII function (ODBC compatible)

Syntax

```
ASCII ( char_expression )
```

Description

The scalar function ASCII returns the ASCII value of the first character of the given character expression.

Example

```
SELECT  ASCII ( zip )
     FROM  customer ;
```

Notes

- The argument to the function must be of type character.
- The result is of type INTEGER.
- If the argument *char_expression* evaluates to null, the result is null.

1.10.2.5 ASIN function (ODBC compatible)

Syntax

```
ASIN ( expression )
```

Description

The scalar function ASIN returns the arcsine of expression.

Example

```
select asin (1) * (180/ pi()) 'Arcsine in degrees' from syscal-
ctable;
ARCSINE IN DEGRE
----------------
90.000000000000000
1 record selected
 select asin (1) 'Arcsine in radians' from syscalctable;
ARCSINE IN RADIA
----------------
1.570796326794897
1 record selected
```

Notes

ASIN takes the ratio (expression) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.

The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

- expression must be in the range -1 to 1.
- expression must evaluate to an approximate numeric data type.

1.10.2.6 ATAN function (ODBC compatible)

Syntax

```
ATAN ( expression )
```

Description

The scalar function ATAN returns the arctangent of expression.

Example

```
select atan (1) * (180/ pi()) 'Arctangent in degrees' from
syscalctable;
ARCTANGENT IN DE
----------------
45.000000000000000
1 record selected
select atan (1) 'Arctangent in radians' from syscalctable;
ARCTANGENT IN RA
----------------
0.785398163397448
```

```
1 record selected
```

Notes

ATAN takes the ratio (expression) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

- expression must be in the range -1 to 1.
- expression must evaluate to an approximate numeric data type.

1.10.2.7 ATAN2 function (ODBC compatible)

Syntax

```
ATAN2 ( expression1 , expression2 )
```

Description

The scalar function ATAN2 returns the arctangent of the x and y coordinates specified by *expression1* and *expression2*.

Example

```
select atan2 (1,1) * (180/ pi()) 'Arctangent in degrees' from
syscalctable;
ARCTANGENT IN DE
----------------
45.000000000000000
1 record selected
select atan2 (1,1) 'Arctangent in radians' from syscalctable;
ARCTANGENT IN RA
----------------
0.785398163397448
1 record selected
```

Notes

ATAN2 takes the ratio of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

expression1 and *expression2* specify the x and y coordinates of the end of the hypotenuse opposite the angle.

The result is expressed in radians and is in the range -Pi/2 to Pi/2 radians. To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

Both *expression1* and *expression2* must evaluate to approximate numeric data types.

1.10.2.8 CASE (SQL-92 Compatible)

Syntax

```
case-expr::
searched-case-expr | simple-case-expr
searched-case-expr::
CASE
     WHEN search_condition THEN { result-expr | NULL }
     [ ... ]
     [ ELSE expr | NULL ]
END
simple-case-expr::
CASE primary-expr
     WHEN expr THEN { result-expr | NULL }
     [ ... ]
     [ ELSE expr | NULL ]
END
```

Description

The CASE scalar function is a type of conditional expression. (See the topic on *Conditional Expressions* on page 1-35 for a summary of all the conditional expressions.)

The general form of the CASE scalar function specifies a series of search conditions and associated result expressions. It is called a searched case expression. SQL returns the value specified by the first result expression whose associated search condition evaluates as true. If none of the search conditions evaluate as true, the CASE expression returns a null value (or the value of some other default expression if the CASE expression includes the ELSE clause).

CASE also supports syntax for a shorthand notation, called a simple case expression, for evaluating whether one expression is equal to a series of other expressions.

Notes

- This function is not allowed in a GROUP BY clause
- Arguments to this function cannot be query expressions

Arguments

CASE

The CASE keyword alone, not followed by primary-expr, specifies a searched case expression. It must be followed by one or more WHEN-THEN clauses each that specify a search condition and corresponding expression.

WHEN search_condition THEN { result-expr | NULL }

WHEN clause for searched case expressions. SQL evaluates search condition. If *search_condition* evaluates as true, CASE returns the value specified by result-expr (or null, if the clause specifies THEN NULL).

If *search_condition* evaluates as false, SQL evaluates the next WHEN-THEN clause, if any, or the ELSE clause, if it is specified.

CASE primary-expr

The CASE keyword followed by an expression specifies a simple case expression. In a simple case expression, one or more WHEN-THEN clauses specify two expressions.

A simple case expression can always be expressed as a searched case expression. Consider the following general simple case expression:

```
CASE primary-expr
     WHEN expr1 THEN result-expr1
     WHEN expr2 THEN result-expr2
     ELSE expr3
END
```

The preceding simple case expression is equivalent to the following searched case expression:

```
CASE
     WHEN primary-expr = expr1 THEN result-expr1
     WHEN primary-expr = expr2 THEN result-expr2
     ELSE expr3
END
```

WHEN expr THEN { result-expr | NULL }

WHEN clause for simple case expressions. SQL evaluates expr and compares it with primary-expr specified in the CASE clause. If they are equal, CASE returns the value specified by result-expr (or null, if the clause specifies THEN NULL).

If expr is not equal to primary-expr, SQL evaluates the next WHEN-THEN clause, if any, or the ELSE clause, if it is specified.

[ELSE { expr | NULL }]

In both searched case expressions and simple case expressions, the ELSE clause specifies an optional expression whose value SQL returns if none of the conditions specified in WHEN-THEN clauses were satisfied. If the CASE expression omits the ELSE clause, it is the same as specifying ELSE NULL.

Examples

The following example shows a searched case expression that assigns a label denoting tables as system tables if they begin with the letters *sys*. Note that this example can not be reformulated as a simple case expression, since it specifies a relational operator other than =.

```
SELECT tbl,
```

```
       CASE
          WHEN tbl like 'sys%' THEN 'System Table'
          ELSE 'Not System table'
       END
FROM systables;

TBL                                SEARCHED_CASE(TBLSY
---                                -------------------
systblspaces                       System Table
systables                          System Table
syscolumns                         System Table
sysindexes                         System Table
sysdbauth                          System Table
systabauth                         System Table
syscolauth                         System Table
sysviews                           System Table
syssynonyms                        System Table
sysdblinks                         System Table
sys_keycol_usage                   System Table
sys_ref_constrs                    System Table
sys_chk_constrs                    System Table
sys_tbl_constrs                    System Table
sys_chkcol_usage                   System Table
sysdatatypes                       System Table
syscalctable                       System Table
systblstat                         System Table
```

The following example shows a searched case expression and an equivalent simple case expression.

```
- Searched case expression:
SELECT tbl,
     CASE
     WHEN tbltype = 'S' THEN 'System Table'
     ELSE 'Not System table'
     End
FROM systables;
- Equivalent simple case expression:
SELECT tbl,
     CASE tbltype
     WHEN  'S' THEN 'System Table'
     ELSE 'Not System table'
```

```
    END
FROM systables;
```

1.10.2.9 CAST function (SQL-92 compatible)

Syntax

```
CAST ( { expression | NULL } AS data_type [(length)] )
```

Description

The scalar function CAST converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

The length option for the *data_type* argument specifies the length for conversions to CHAR and VARCHAR data types. If omitted, the default is 30 bytes.

If the expression evaluates to null, the result of the function is null. Specifying NULL with the CAST function is useful for set operations such as UNION that require two table to have the same structure. CAST NULL allows you to specify a column of the correct data type so a table with a similar structure to another, but with fewer columns, can be in a union operation with the other table.

The CAST function provides a data-type-conversion mechanism compatible with the SQL-92 standard.

Use the CONVERT function, enclosed in the ODBC escape clause {fn }, to specify ODBC-compliant syntax for data type conversion. See the CONVERT (ODBC compatible) function for more information.

Example

The following SQL example uses CAST to convert an integer field from a catalog table to a character data type:

```
SELECT CAST(fld AS CHAR(25)), fld  FROM systpe.syscalctable;
CONVERT(CHARACTER(25),FLD)          FLD
--------------------------          ---
100                                 100
1 record selected
```

1.10.2.10 CEILING function (ODBC compatible)

Syntax

```
CEILING ( expression )
```

Description

The scalar function CEILING returns the smallest integer greater than or equal to expression.

Example

```
SELECT CEILING (32.5) 'Ceiling'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression must evaluate to a numeric data type.

1.10.2.11 CHAR function (ODBC compatible)

Syntax

```
CHAR ( integer_expression )
```

Description

The scalar function CHAR returns a character string with the first character having an ASCII value equal to the argument expression. CHAR is identical to CHR but provides ODBC-compatible syntax.

Example

```
SELECT *
     FROM   customer
     WHERE  SUBSTR (zip, 1, 1) = CHAR (53) ;
```

Notes

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type character.
- If the argument *integer_expression* evaluates to null, the result is null.

1.10.2.12 CHARTOROWID (extension)

Syntax

```
CHARTOROWID ( char_expression )
```

Description

The scalar function CHARTOROWID returns a ROWID contained in the input argument in character form. The representation of a row identifier depends on the storage manager. The format of the *char_expression* argument to this function varies between storage managers.

Example

The following example shows the character-string format for a row identifier supplied as an argument to CHARTOROWID. In this example, the format for a row identifier is an integer (delimited as a character string by single quotes).

```
SELECT ROWID, FLD FROM SYSCALCTABLE;
```

```
ROWID                          FLD
-----                          ---
0                              100
1 record selected
-- CHARTOROWID requires single quotes around its argument
SELECT * FROM SYSCALCTABLE WHERE ROWID = CHARTOROWID ('0');
         FLD
         ---
         100
1 record selected
```

Notes

- The argument to the function must be of type character.
- The result is of internal ROWID type as defined by the storage manager.
- If the argument *char_expression* evaluates to null, the result is null.
- The SQL statement execution returns error if the result of the input character expression does not contain a character string in the proper format for a row identifier, as defined by the storage manager.

1.10.2.13 CHR function (extension)

Syntax

```
CHR ( integer_expression )
```

Description

The scalar function CHR returns a character string with the first character having an ASCII value equal to the argument expression.

Example

```
SELECT *
     FROM   customer
     WHERE  SUBSTR (zip, 1, 1) = CHR (53) ;
```

Notes

- The argument to the function must be of type INTEGER, TINYINT, or SMALLINT.
- The result is of type character.
- If the argument *integer_expression* evaluates to null, the result is null.

1.10.2.14 COALESCE (SQL-92 compatible)

Syntax

```
COALESCE ( expression1, expression2 [ , ... ] )
```

Description

The COALESCE scalar function is a type of conditional expression. (See the topic on *Conditional Expressions* on page 1-35 for a summary of all the conditional expressions.)

COALESCE specifies a series of expressions, and returns the first expression whose value is not null. If all the expressions evaluate as null, COALESCE returns a null value.

The COALESCE syntax is shorthand notation for a common case that can also be represented in a CASE expression. The following two formulations are equivalent:

```
COALESCE ( expression1 , expression2 , expression3 )
CASE
     WHEN expression1 IS NOT NULL THEN expression1
     WHEN expression2 IS NOT NULL THEN expression2
     ELSE expression3
```

Example

```
SELECT COALESCE(end_date, start_date) from job_hist;
```

Notes

- This function is not allowed in a GROUP BY clause
- Arguments to this function cannot be query expressions

1.10.2.15 CONCAT function (ODBC compatible)

Syntax

```
CONCAT ( char_expression , char_expression )
```

Description

The scalar function CONCAT returns a concatenated character string formed by concatenating argument one with argument two.

The CONCAT scalar function is similar to the concatenation operator. However, the concatenation operator allows easy concatenation of more than two character expressions, while the CONCAT function requires nesting.

Example

```
SELECT name, empno, salary
     FROM customer
     WHERE project = CONCAT('US',proj_nam);
```

Notes

- Both the arguments must be of type CHARACTER or VARCHAR.
- The result is of type VARCHAR.
- If any of the argument expressions evaluates to null, the result is null.
- The trailing blanks for the first argument are removed.

1.10.2.16 CONVERT function (extension)

Syntax

```
CONVERT ( 'data_type[(length)]', expression )
```

Description

The scalar function CONVERT converts an expression to another data type. The first argument is the target data type. The second argument is the expression to be converted to that type.

The length option for the *data_type* argument specifies the length for conversions to CHAR and VARCHAR data types. If omitted, the default is 30 bytes.

If the expression evaluates to null, the result of the function is null.

The CONVERT function syntax is similar to but not compatible with the ODBC CONVERT function. Enclose the function in the ODBC escape clause {fn }, to specify ODBC-compliant syntax. See the CONVERT function (ODBC compatible) topic for more information.

Example

The following SQL example converts an integer field from a catalog table to a character string:

```
SELECT CONVERT('CHAR', fld), fld  FROM systpe.syscalctable;
CONVERT(CHAR,FLD)                          FLD
-----------------                          ---
100                                        100
1 record selected
SELECT CONVERT('CHAR(35)', fld), fld
     FROM systpe.syscalctable;
CONVERT(CHAR(35),FLD)                          FLD
---------------------                          ---
100                                            100
1 record selected
```

1.10.2.17 CONVERT function (ODBC compatible)

Syntax

```
{fn CONVERT (expression , data_type ) }

data_type::
     SQL_BIGINT
|    SQL_BINARY
|    SQL_BIT
|    SQL_CHAR
|    SQL_DATE
|    SQL_DECIMAL
|    SQL_DOUBLE
|    SQL_FLOAT
|    SQL_INTEGER
|    SQL_LONGVARBINARY
|    SQL_LONGVARCHAR
|    SQL_REAL
|    SQL_SMALLINT
|    SQL_TIME
|    SQL_TIMESTAMP
|    SQL_TINYINT
|    SQL_VARBINARY
|    SQL_VARCHAR
```

Description

The ODBC scalar function CONVERT converts an expression to another data type. The first argument is the expression to be converted. The second argument is the target data type.

If the expression evaluates to null, the result of the function is null.

The ODBC CONVERT function provides ODBC-compliant syntax for data type conversion. You must enclose the function with the ODBC escape clause {fn } to use ODBC-compliant syntax.

1.10.2.18 COS function (ODBC compatible)

Syntax

```
COS ( expression )
```

Description

The scalar function COS returns the cosine of expression.

Example

```
select cos(45 * pi()/180) 'Cosine of 45 degrees' from syscal-
ctable;
COSINE OF 45 DEG
----------------
0.707106781186548
1 record selected
```

Notes

COS takes an angle (expression) and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse.

- expression specifies an angle in radians
- expression must evaluate to an approximate numeric data type.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

1.10.2.19 CURDATE function (ODBC compatible)

Syntax

```
CURDATE ()
```

Description

CURDATE returns the current date as a DATE value. This function takes no arguments.

SQL statements can refer to CURDATE anywhere they can refer to a DATE expression. For example,

```
INSERT INTO objects (object_owner, object_id, create_date)
           VALUES (USER, 1001, CURDATE()) ;
```

1.10.2.20 CURTIME function (ODBC compatible)

Syntax

```
CURTIME ()
```

Description

CURTIME returns the current time as a TIME value. This function takes no arguments.

SQL statements can refer to CURTIME anywhere they can refer to a TIME expression. For example,

```
INSERT INTO objects (object_owner, object_id, create_time)
           VALUES (USER, 1001, CURTIME()) ;
```

1.10.2.21 DATABASE (ODBC compatible)

Syntax

```
DATABASE [ ( ) ]
```

Description

The scalar function DATABASE returns the name of the database corresponding to the current connection name. This function takes no arguments, and the trailing parentheses are optional.

Example

```
select database() from t2;
DATABASE
--------
steel
1 record selected
```

1.10.2.22 DAYNAME function (ODBC compatible)

Syntax

```
DAYNAME ( date_expression )
```

Description

Returns a character string containing the name of the day (for example, Sunday, through Saturday) for the day portion of *date_expression*. The argument *date_expression* can be the name of a column, the result of another scalar function, or a date or timestamp literal.

Example

```
SELECT *
FROM orders
WHERE order_no = 342 and DAYNAME(order_date)='SATURDAY';


    ORDER_NO ORDER_DATE   REFERENCE                    CUST_NO
 342      08/10/1991      tdfg/101                          10001


1 record selected
```

1.10.2.23 DAYOFMONTH function (ODBC compatible)

Syntax

```
DAYOFMONTH ( date_expression )
```

Description

The scalar function DAYOFMONTH returns the day of the month in the argument as a short integer value in the range of 1 - 31.

Example

```
SELECT *
     FROM  orders
     WHERE DAYOFMONTH (order_date) = 14 ;
```

Notes

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.24 DAYOFWEEK function (ODBC compatible)

Syntax

```
DAYOFWEEK ( date_expression )
```

Description

The scalar function DAYOFWEEK returns the day of the week in the argument as a short integer value in the range of 1 - 7.

Example

```
SELECT *
     FROM  orders
     WHERE DAYOFWEEK (order_date) = 2 ;
```

Notes

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.25 DAYOFYEAR function (ODBC compatible)

Syntax

```
DAYOFYEAR ( date_expression )
```

Description

The scalar function DAYOFYEAR returns the day of the year in the argument as a short integer value in the range of 1 - 366.

Example

```
SELECT *
     FROM  orders
     WHERE DAYOFYEAR (order_date) = 300 ;
```

Notes

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.26 DB_NAME (extension)

Syntax

```
DB_NAME ( )
```

Description

The scalar function DB_NAME returns the name of the database corresponding to the current connection name. It provides compatibility with the Sybase SQL Server function *db_name.*

Example

```
SELECT DB_NAME() FROM T2;
DB_NAME
-------
dharmav4
1 record selected
```

1.10.2.27 DECODE function (extension)

Syntax

```
DECODE ( expression, search_expression, match_expression
               [ , search_expression, match_expression ...]
               [ , default_expression ] )
```

Description

The DECODE scalar function is a type of conditional expression. (See the topic on *Conditional Expressions* on page 1-35 for a summary of all the conditional expressions.)

The scalar function DECODE compares the value of the first argument expression with each *search_expression* and if a match is found, returns the corresponding *match_expression*. If no match is found, then the function returns *default_expression*. If *default_expression* is not specified and no match is found, the function returns a null value.

DECODE provides a subset of the functionality of CASE that is compatible with Oracle SQL syntax. Use a simple case expression for SQL-compatible syntax (see the CASE function on page 1-50).

Example

```
SELECT ename, DECODE (deptno,
                          10, 'ACCOUNTS    ',
                          20, 'RESEARCH    ',
                          30, 'SALES       ',
                          40, 'SUPPORT     ',
                          'NOT ASSIGNED'
                     )
     FROM employee ;
```

Notes

- The first argument expression can be of any type. The types of all *search_expressions* must be compatible with the type of the first argument.
- The *match_expressions* can be of any type. The types of all *match_expressions* must be compatible with the type of the first *match_expression*.
- The type of the *default_expression* must be compatible with the type of the first *match_expression*.
- The type of the result is the same as that of the first *match_expression*.
- If the first argument expression is null then the value of the *default_expression* is returned, if it is specified. Otherwise null is returned.

1.10.2.28 DEGREES function (ODBC compatible)

Syntax

```
DEGREES ( expression )
```

Description

The scalar function DEGREES returns the number of degrees in an angle specified in radians by expression.

Example

```
SELECT DEGREES(3.14159265359) 'Degrees in pi Radians'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression specifies an angle in radians
- expression must evaluate to a numeric data type.

1.10.2.29 DIFFERENCE function (ODBC compatible)

Syntax

```
DIFFERENCE ( string_exp1,string_exp2 )
```

Description

The scalar function DIFFERENCE returns an integer value that indicates the difference between the values returned by the SOUNDEX function for *string_exp1* and *string_exp2*.

Example

```
SELECT DIFFERENCE(name,'Robets')
FROM customer
WHERE  name = 'Roberts';

DIFFEREN

       2
1 record selected
```

Notes

- The arguments of the function can be of the type fixed length or variable length CHARACTER.
- The result is INTEGER.
- If the argument expression evaluates to null, the result is null.

1.10.2.30 EXP function (ODBC compatible)

Syntax

```
EXP ( expression )
```

Description

The scalar function EXP returns the exponential value of expression (e raised to the power of expression).

Example

```
SELECT EXP( 4 ) 'e to the 4th power'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression must evaluate to an approximate numeric data type.

1.10.2.31 FLOOR function (ODBC compatible)

Syntax

```
FLOOR ( expression )
```

Description

The scalar function FLOOR returns the largest integer less than or equal to expression.

Example

```
SELECT FLOOR (32.5) 'Floor'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression must evaluate to a numeric data type.

1.10.2.32 GREATEST function (extension)

Syntax

```
GREATEST ( expression, expression, ... )
```

Description

The scalar function GREATEST returns the greatest value among the values of the given expressions.

Example

```
SELECT  cust_no, name,
     GREATEST (ADD_MONTHS (start_date, 10), SYSDATE)
     FROM  customer ;
```

Notes

- The first argument to the function can be of any type. The types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.33 HOUR function (ODBC compatible)

Syntax

```
HOUR ( time_expression )
```

Description

The scalar function HOUR returns the hour in the argument as a short integer value in the range of 0 - 23.

Example

```
SELECT *
     FROM  arrivals
     WHERE HOUR (in_time) < 12 ;
```

Notes

- The argument to the function must be of type TIME.
- The argument must be specified in the format hh:mi:ss.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.34 IFNULL function (ODBC compatible)

Syntax

```
IFNULL( expr, value)
```

Description

The scalar function IFNULL returns value if expr is null. If expr is not null, IFNULL returns expr.

Example

```
select c1, ifnull(c1, 9999) from temp order by c1;
c1     ifnull(c1,9999)
       9999
       9999
       9999
1      1
3      3
```

Notes

The data type of value must be compatible with the data type of expr.

1.10.2.35 INITCAP function (extension)

Syntax

```
INITCAP ( char_expression )
```

Description

The scalar function INITCAP returns the result of the argument character expression after converting the first character to upper case and the subsequent characters to lower case.

Example

```
SELECT INITCAP (name)
     FROM  customer ;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

1.10.2.36 INSERT function (ODBC compatible)

Syntax

```
INSERT(string_exp1,start,length,string_exp2)
```

Description

The scalar function INSERT returns a character string where length characters have been deleted from *string_exp1* beginning at start and *string_exp2* has been inserted into *string_exp1*, beginning at start.

Example

```
SELECT INSERT(name,2,4,'xx')
FROM customer
WHERE name = 'Goldman';
INSERT(NAME,2,4,XX)

Gxxan
1 record selected
```

Notes

- The *string_exp* can be of the type fixed length or variable length CHARACTER.
- The start and length can be of the type INTEGER, SMALLINT, TINYINT or BIGINT.
- The result string is of the type *string_exp1*.
- If any of the argument expression evaluates to a null, the result would be a null.
- If start is negative or zero, the result string evaluates to a null.
- If length is negative, the result evaluates to a null.

1.10.2.37 INSTR function (extension)

Syntax

```
INSTR ( char_expression, char_expression
                  [, start_position [, occurrence]])
```

Description

The scalar function INSTR searches for the character string corresponding to the second argument in the character string corresponding to the first argument starting at *start_position*. If *occurrence* is specified, then INSTR searches for the *nth* occurrence where *n* is the value of the fourth argument.

The position (with respect to the start of string corresponding to the first argument) is returned if a search is successful. Zero is returned if no match can be found.

Example

```
SELECT cust_no, name
     FROM  customer
     WHERE INSTR (LOWER (addr), 'heritage') > 0 ;
```

Notes

- The first and second arguments must be of type CHARACTER.
- The third and fourth arguments, if specified, must be of type INTEGER.
- The values for specifying position in a character string starts from 1. That is, the very first character in a string is at position 1, the second character is at position 2 and so on.
- If the third argument is not specified, a default value of 1 is assumed.
- If the fourth argument is not specified, a default value of 1 is assumed.
- The result is of type INTEGER.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.38 LAST_DAY function (extension)

Syntax

```
LAST_DAY ( date_expression )
```

Description

The scalar function LAST_DAY returns the date corresponding to the last day of the month containing the argument date.

Example

```
SELECT *
     FROM  orders
```

```
WHERE LAST_DAY (order_date) + 1 = '08/01/1991' ;
```

Notes

- The argument to the function must be of type DATE.
- The result is of type DATE.
- If the argument expression evaluates to null, the result is null.

1.10.2.39 LCASE function (ODBC compatible)

Syntax

```
LCASE ( char_expression )
```

Description

The scalar function LCASE returns the result of the argument character expression after converting all the characters to lower case. LCASE is the same as LOWER but provides ODBC-compatible syntax.

Example

```
SELECT *
     FROM  customer
     WHERE LCASE (name) = 'smith' ;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

1.10.2.40 LEAST function (extension)

Syntax

```
LEAST ( expression, expression, ... )
```

Description

The scalar function LEAST returns the lowest value among the values of the given expressions.

Example

```
SELECT cust_no, name,
     LEAST (ADD_MONTHS (start_date, 10), SYSDATE)
     FROM  customer ;
```

Notes

- The first argument to the function can be of any type. The types of the subsequent arguments must be compatible with that of the first argument.
- The type of the result is the same as that of the first argument.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.41 LEFT function (ODBC compatible)

Syntax

```
LEFT ( string_exp, count )
```

Description

The scalar function LEFT returns the leftmost count of characters of *string_exp*.

Example

```
SELECT LEFT(name,4)
FROM   customer
WHERE  name = 'Goldman';

LEFT(NAME,4)

Gold
1 record selected
```

Notes

- The *string_exp* can be of the type fixed or variable length CHARACTER.
- The count can be of the type INTEGER, SMALLINT, BIGINT, or TINYINT.
- If any of the arguments of the expression evaluates to a null, the result would be null.
- If the count is negative, the result evaluates to a null.

1.10.2.42 LENGTH function (ODBC compatible)

Syntax

```
LENGTH ( char_expression )
```

Description

The scalar function LENGTH returns the string length of the value of the given character expression.

Example

```
SELECT name 'LONG NAME'
```

```
FROM  customer
WHERE LENGTH (name) > 5 ;
```

Notes

- The argument to the function must be of type CHARACTER or VARCHAR.
- The result is of type INTEGER.
- If the argument expression evaluates to null, the result is null.

1.10.2.43 LOCATE function (ODBC compatible)

Syntax

```
LOCATE( char-expr1 , char-expr2, [start-position] )
```

Description

The scalar function LOCATE returns the location of the first occurrence of char-expr1 in char-expr2. If the function includes the optional integer argument start-position, LOCATE begins searching char-expr2 at that position. If the function omits the start-position argument, LOCATE begins its search at the beginning of char-expr2.

LOCATE denotes the first character position of a character expression as 1. If the search fails, LOCATE returns 0. If either character expression is null, LOCATE returns a null value.

Example

The following example uses two string literals as character expressions. LOCATE returns a value of 6:

```
SELECT  LOCATE('this', 'test this test', 1) FROM TEST;
LOCATE(THIS,
------------
           6
1 record selected
```

1.10.2.44 LOG10 function (ODBC compatible)

Syntax

```
LOG10 ( expression )
```

Description

The scalar function LOG10 returns the base 10 logarithm of expression.

Example

```
SELECT LOG10 (100) 'Log base 10 of 100'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression must evaluate to an approximate numeric data type.

1.10.2.45 LOWER function (SQL-92 compatible)

Syntax

```
LOWER ( char_expression )
```

Description

The scalar function LOWER returns the result of the argument character expression after converting all the characters to lower case.

Example

```
SELECT *
     FROM  customer
     WHERE LOWER (name) = 'smith' ;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

1.10.2.46 LPAD function (extension)

Syntax

```
LPAD ( char_expression, length [, pad_expression] )
```

Description

The scalar function LPAD pads the character string corresponding to the first argument on the left with the character string corresponding to the third argument so that after the padding, the length of the result is length.

Example

```
SELECT LPAD (name, 30)
     FROM customer ;
SELECT LPAD (name, 30, '.')
       FROM customer ;
```

Notes

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type INTEGER.
- The third argument, if specified, must be of type CHARACTER.

- If the third argument is not specified, the default value is a string of length 1 containing one blank.
- If L1 is the length of the first argument and L2 is the value of the second argument, then:
  - If L1 is less than L2, the number of characters padded is equal to L2 - L1.
  - If L1 is equal to L2, no characters are padded and the result string is the same as the first argument.
  - If L1 is greater than L2, the result string is equal to the first argument truncated to first L2 characters.

The result is of type CHARACTER.

- If the argument expression evaluates to null, the result is null.

1.10.2.47 LTRIM function (ODBC compatible)

Syntax

```
LTRIM ( char_expression [ , char_set ] )
```

Description

The scalar function LTRIM removes all the leading characters in *char_expression*, that are present in *char_set* and returns the resultant string. Thus, the first character in the result is guaranteed to be not in *char_set*. If the *char_set* argument is omitted, the function removes the leading and trailing blanks from *char_expression*.

Example

```
SELECT name, LTRIM (addr, ' ')
     FROM  customer ;
```

Notes

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

1.10.2.48 MINUTE function (ODBC compatible)

Syntax

```
MINUTE ( time_expression )
```

Description

The scalar function MINUTE returns the minute value in the argument as a short integer in the range of 0 - 59.

Example

```
SELECT *
     FROM  arrivals
     WHERE MINUTE (in_time) > 10 ;
```

Notes

- The argument to the function must be of type TIME.
- The argument must be specified in the format HH:MI:SS.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.49 MOD function (ODBC compatible)

Syntax

```
MOD ( expression1, expression2 )
```

Description

The scalar function MOD returns the remainder of *expression1* divided by *expression2*.

Example

```
SELECT MOD (11, 4) 'Modulus'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- Both *expression1* and *expression2* must evaluate to exact numeric data types.
- If *expression2* evaluates to zero, MOD returns zero.

1.10.2.50 MONTHNAME function (ODBC compatible)

Syntax

```
MONTHNAME ( date_expression )
```

Description

Returns a character string containing the name of the month (for example, January, through December) for the month portion of *date_expression.* Argument *date_expression* can be name of a column, the result of another scalar function, or a date or timestamp literal.

Example

```
SELECT *
FROM orders
WHERE order_no =346 and MONTHNAME(order_date)='JUNE';
```

```
    ORDER_NO ORDER_DATE   REFERENCE              CUST_NO

 346    06/01/1991        87/rd                        10002
 1 record selected
```

1.10.2.51 MONTH function (ODBC compatible)

Syntax

```
MONTH ( date_expression )
```

Description

The scalar function MONTH returns the month in the year specified by the argument as a short integer value in the range of 1 - 12.

Example

```
SELECT *
     FROM  orders
     WHERE MONTH (order_date) = 6 ;
```

Notes

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.52 MONTHS_BETWEEN function (extension)

Syntax

```
MONTHS_BETWEEN ( date_expression, date_expression )
```

Description

The scalar function MONTHS_BETWEEN computes the number of months between two date values corresponding to the first and second arguments.

Example

```
SELECT MONTHS_BETWEEN (SYSDATE, order_date)
     FROM  orders
     WHERE order_no = 1002 ;
```

Notes

- The first and the second arguments to the function must be of type DATE.
- The result is of type INTEGER.

- The result is negative if the date corresponding to the second argument is greater than that corresponding to the first argument.
- If any of the arguments expression evaluates to null, the result is null.

1.10.2.53 NEXT_DAY function (extension)

Syntax

```
NEXT_DAY ( date_expression, day_of_week )
```

Description

The scalar function NEXT_DAY returns the minimum date that is greater than the date corresponding to the first argument for which the day of the week is same as that specified by the second argument.

Example

```
SELECT NEXT_DAY (order_date, 'MONDAY')
     FROM  orders ;
```

Notes

- The first argument to the function must be of type DATE.
- The second argument to the function must be of type CHARACTER. The result of the second argument must be a valid day of week ('SUNDAY', 'MONDAY' etc.)
- The result is of type DATE.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.54 NOW function (ODBC compatible)

Syntax

```
NOW ( )
```

Description

NOW returns the current date and time as a TIMESTAMP value. This function takes no arguments.

1.10.2.55 NULLIF (SQL-92 compatible)

Syntax

```
NULLIF ( expression1, expression2 )
```

Description

The NULLIF scalar function is a type of conditional expression. (See the topic on *Conditional Expressions* on page 1-35 for a summary of all the conditional expressions.)

The NULLIF scalar function returns a null value for *expression1* if it is equal to *expression2*. It's useful for converting values to null from applications that use some other representation for missing or unknown data.

Notes

- This function is not allowed in a GROUP BY clause.
- Arguments to this function cannot be query expressions.
- The NULLIF expression is shorthand notation for a common case that can also be represented in a CASE expression, as follows:

```
CASE
     WHEN expression1 = expression2 THEN NULL
     ELSE expression1
```

Example

This example uses the NULLIF scalar function to insert a null value into an address column if the host-language variable contains a single space character.

```
INSERT INTO employee (add1) VALUES (NULLIF (:address1, ' '));
```

1.10.2.56 NVL function (extension)

Syntax

```
NVL ( expression, expression )
```

Description

The scalar function NVL returns the value of the first expression if the first expression value is not null. If the first expression value is null, the value of the second expression is returned.

The NVL function is not ODBC compatible. Use the IFNULL function for ODBC-compatible syntax.

Example

```
SELECT salary + NVL (comm, 0) 'TOTAL SALARY'
     FROM  employee ;
```

Notes

- The first argument to the function can be of any type.
- The type of the second argument must be compatible with that of the first argument.
- The type of the result is the same as the first argument.

1.10.2.57 OBJECT_ID function (extension)

Syntax

```
OBJECT_ID ('table_name')
```

Description

The scalar function OBJECT_ID returns the value of the id column in the *systpe.systables*, plus one. This function provides compatibility with the Sybase SQL Server function *object_id*.

Arguments

table_name
The name of the table for which OBJECT_ID returns an identification value.

Example

```
select id, object_id(tbl), tbl from systpe.systables
   1  where owner = 'systpe';
          ID OBJECT_ID(TB TBL
          -- ------------ ---
           0            1 systblspaces
           1            2 systables
           2            3 syscolumns
           3            4 sysindexes
           4            5 systsfiles
           5            6 syslogfiles
           6            7 sysdbbackup
           7            8 syslogbackup
           8            9 sysdbsyncpt
           9           10 sysdbsuuid
          10           11 syssyssvr
          11           12 sysusrsvr
.
.
.
```

1.10.2.58 PI function (ODBC compatible)

Syntax

```
PI ( )
```

Description

The scalar function PI returns the constant value of pi as a floating point value.

Example

```
SELECT PI ()
     FROM SYSTPE.SYSCALCTABLE;
```

1.10.2.59 POWER function (ODBC compatible)

Syntax

```
POWER ( expression1 , expression2 )
```

Description

The scalar function POWER returns *expression1* raised to the power of *expression2*.

Example

```
SELECT POWER ( 3 , 2) '3 raised to the 2nd power'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- *expression1* must evaluate to a numeric data type.
- *expression2* must evaluate to an exact numeric data type.

1.10.2.60 PREFIX function (extension)

Syntax

```
PREFIX(char_expression, start_position, char_expression)
```

Description

The scalar function PREFIX returns the substring of a character string starting from the position specified by start position, and ending before the specified character.

Arguments

char_expression
An expression that evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to null, PREFIX returns null.

start_position
An expression that evaluates to an integer value. PREFIX searches the string specified in the first argument starting at that position. A value of 1 indicates the first character of the string.

char_expression
An expression that evaluates to a single character. PREFIX returns the substring that ends before that character. If PREFIX does not find the character, it returns the substring beginning with *start_position*, to the end of the string. If the expression evaluates to more than one character, PREFIX ignores all but the first character.

Example

```
SELECT C1, C2, PREFIX(C1, 1, '.') FROM T1;
C1           C2  PREFIX(C1,1,.
--           --  -------------
test.pref    .   test
pref.test    s   pref
2 records selected
SELECT C1, C2, PREFIX(C1, 1, C2) FROM T1;
C1           C2  PREFIX(C1,1,C
--           --  -------------
test.pref    .   test
pref.test    s   pref.te
2 records selected
SELECT C1, C2, PREFIX(C1, 1, 'Q') FROM T1;
C1           C2  PREFIX(C1,1,Q
--           --  -------------
test.pref    .   test.pref
pref.test    s   pref.test
2 records selected
```

1.10.2.61 QUARTER function (ODBC compatible)

Syntax

```
QUARTER ( time_expression )
```

Description

The scalar function QUARTER returns the quarter in the year specified by the argument as a short integer value in the range of 1 - 4.

Example

```
SELECT *
     FROM  orders
     WHERE QUARTER (order_date) = 3 ;
```

Notes

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.62 RADIANS function (ODBC compatible)

Syntax

```
RADIANS ( expression )
```

Description

The scalar function RADIANS returns the number of radians in an angle specified in degrees by expression.

Example

```
SELECT RADIANS(180) 'Radians in 180 degrees'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression specifies an angle in degrees
- expression must evaluate to a numeric data type.

1.10.2.63 RAND function (ODBC compatible)

Syntax

```
RAND ( [ expression ] )
```

Description

The scalar function RAND returns a randomly-generated number, using expression as an optional seed value.

Example

```
SELECT RAND(3) 'Random number using 3 as seed value'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression must evaluate to an exact numeric data type.

1.10.2.64 REPLACE function (ODBC compatible)

Syntax

```
REPLACE ( string_exp1,string_exp2,string_exp3 )
```

Description

The scalar function REPLACE replaces all occurrences of *string_exp2* in *string_exp1* with *string_exp3*.

Example

```
SELECT REPLACE ( name,'mi','moo' )
```

```
FROM customer
WHERE name = 'Smith';

REPLACE(NAME,MI,MOO)

Smooth
1 record selected
```

Notes

- *string_exp* can be of the type fixed or variable length CHARACTER.
- If any of the arguments of the expression evaluates to null, the result is null.
- If the replacement string is not found in the search string, it returns the original string.

1.10.2.65 RIGHT function (ODBC compatible)

Syntax

```
RIGHT ( string_exp, count )
```

Description

The scalar function RIGHT returns the rightmost count of characters of *string_exp*.

Example

```
SELECT RIGHT(fld1,6)
FROM   test100
WHERE  fld1 = 'Afghanistan';
RIGHT(FLD1,6)

nistan
1 record selected
```

Notes

- The *string_exp* can be of the type fixed or variable length CHARACTER.
- The count can be of the type INTEGER, SMALLINT, BIGINT, or TINYINT.
- If any of the arguments of the expression evaluates to a null, the result would be null.
- If the count is negative, the result evaluates to a null.

1.10.2.66 REPEAT function (ODBC compatible)

Syntax

```
REPEAT ( string_exp,count )
```

Description

The scalar function REPEAT returns a character string composed of *string_exp* repeated count times.

Example

```
SELECT REPEAT(fld1,3)
FROM   test100
WHERE  fld1 = 'Afghanistan'

Results

REPEAT(FLD1,3)

AfghanistanAfghanistanAfghanistan

1 record selected
```

Notes

- The *string_exp* can be of the type fixed length or variable length CHARACTER.
- The count can be of the type INTEGER, SMALLINT, BIGINT, or TINYINT.
- If any of the arguments of the expression evaluates to a null, the result would be null.
- If the count is negative or zero, the result evaluates to a null.

1.10.2.67 ROWID (extension)

Syntax

```
ROWID
```

Description

ROWID returns the row identifier of the current row in a table. This function takes no arguments. The ROWID of a row is determined when the row is inserted into the table. Once assigned, the ROWID remains the same for the row until the row is deleted. At any given time, each row in a table is uniquely identified by its ROWID.

The format of the row identifier returned by this function varies between storage managers.

Selecting a row in a table using its ROWID is the most efficient way of selecting the row. For example,

```
SELECT *
FROM customers
WHERE ROWID = '10';
```

1.10.2.68 ROWIDTOCHAR (extension)

Syntax

```
ROWIDTOCHAR ( expression )
```

Description

The scalar function ROWIDTOCHAR returns the character form of a ROWID contained in the input argument. The representation of a row identifier depends on the storage manager. The format of the argument to this function is defined by the storage manager. See the documentation for your storage manager for details.

Example

The following example uses ROWIDTOCHAR to convert a row identifier from its internal representation to a character string. This example is specific to the Dharma storage manager's representation of a row identifier:

```
SELECT cust_no,
   SUBSTR (ROWIDTOCHAR (ROWID),  1, 8) 'PAGE NUMBER',
   SUBSTR (ROWIDTOCHAR (ROWID), 10, 4) 'LINE NUMBER',
   SUBSTR (ROWIDTOCHAR (ROWID), 15, 4) 'TABLE SPACE NUMBER'
     FROM customer ;
```

Notes

- The argument to the function must be a ROWID, as defined by the storage manager.
- The result is of CHARACTER type.
- If the argument expression evaluates to null, the result is null.

1.10.2.69 RPAD function (extension)

Syntax

```
RPAD ( char_expression, length [, pad_expression] )
```

Description

The scalar function RPAD pads the character string corresponding to the first argument on the right with the character string corresponding to the third argument so that after the padding, the length of the result would be equal to the value of the second argument length.

Example

```
SELECT RPAD (name, 30)
     FROM customer ;

     SELECT RPAD (name, 30, '.')
     FROM customer ;
```

Notes

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type INTEGER.
- The third argument, if specified, must be of type CHARACTER.
- If the third argument is not specified, the default value is a string of length 1 containing one blank.
- If L1 is the length of the first argument and L2 is the value of the second argument, then:
  - If L1 is less than L2, the number of characters padded is equal to L2 - L1.
  - If L1 is equal to L2, no characters are padded and the result string is the same as the first argument.
  - If L1 is greater than L2, the result string is equal to the first argument truncated to first L2 characters.

The result is of type CHARACTER.

- If the argument expression evaluates to null, the result is null.

1.10.2.70 RTRIM function (ODBC compatible)

Syntax

```
RTRIM ( char_expression [ , char_set ] )
```

Description

The scalar function RTRIM removes all the trailing characters in *char_expression*, that are present in *char_set* and returns the resultant string. Thus, the last character in the result is guaranteed to be not in *char_set*. If the *char_set* argument is omitted, the function removes the leading and trailing blanks from *char_expression*.

Example

```
SELECT RPAD ( RTRIM (addr, ' '), 30, '.')
     FROM customer ;
```

Notes

- The first argument to the function must be of type CHARACTER.
- The second argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

1.10.2.71 SECOND function (ODBC compatible)

Syntax

```
SECOND ( time_expression )
```

Description

The scalar function SECOND returns the seconds in the argument as a short integer value in the range of 0 - 59.

Example

```
SELECT *
     FROM  arrivals
     WHERE SECOND (in_time) <= 40 ;
```

Notes

- The argument to the function must be of type TIME.
- The argument must be specified in the format HH:MI:SS.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.72 SIGN function (ODBC compatible)

Syntax

```
SIGN ( expression )
```

Description

The scalar function SIGN returns 1 if expression is positive, -1 if expression is negative, or zero if it is zero.

Example

```
SELECT SIGN(-14) 'Sign'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- expression must evaluate to a numeric data type.

1.10.2.73 SIN function (ODBC compatible)

Syntax

```
SIN ( expression )
```

Description

The scalar function SIN returns the sine of expression.

Example

```
select sin(45 * pi()/180) 'Sine of 45 degrees' from syscal-
ctable;
SINE OF 45 DEGRE
----------------
0.707106781186547
1 record selected
```

Notes

SIN takes an angle (expression) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse.

- expression specifies an angle in radians
- expression must evaluate to an approximate numeric data type.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

1.10.2.74 SOUNDEX function (ODBC compatible)

Syntax

```
SOUNDEX ( string_exp )
```

Description

The scalar function SOUNDEX returns a four-character soundex code for character strings that are composed of a contiguous sequence of valid single- or double-byte roman letters.

Example

```
SELECT SOUNDEX('Roberts')
FROM syscalctable;
```

1.10.2.75 SPACE function (ODBC compatible)

Syntax

```
SPACE ( count )
```

Description

The scalar function SPACE returns a character string consisting of count spaces.

Example

```
SELECT CONCAT(SPACE(3), name)
FROM customer
WHERE name = 'Roberts';
```

```
CONCAT (   ,NAME)

   Roberts
1 record selected
```

Notes

- The count argument can be of type INTEGER, SMALLINT, BIGINT, or TINYINT.
- If count is null, the result is null.
- If count is negative, the result is null.

1.10.2.76 SQRT function (ODBC compatible)

Syntax

```
SQRT ( expression )
```

Description

The scalar function SQRT returns the square root of expression.

Example

```
SELECT SQRT(28) 'square root of 28'
     FROM SYSTPE.SYSCALCTABLE;
```

Notes

- The value of expression must be positive.
- expression must evaluate to an approximate numeric data type.

1.10.2.77 SUBSTR function (extension)

Syntax

```
SUBSTR ( char_expression, start_position [, length ] )
```

Description

The scalar function SUBSTR returns the substring of the character string corresponding to the first argument starting at *start_position* and length characters long. If the third argument length is not specified, substring starting at *start_position* up to the end of *char_expression* is returned.

Example

```
SELECT name, '(', SUBSTR (phone, 1, 3) , ')',
               SUBSTR (phone, 4, 3), '-',
               SUBSTR (phone, 7, 4)
```

```
    FROM customer ;
```

Notes

- The first argument must be of type CHARACTER.
- The second argument must be of type INTEGER.
- The third argument, if specified, must be of type INTEGER.
- The values for specifying position in the character string start from 1: The very first character in a string is at position 1, the second character is at position 2 and so on.
- The result is of type CHARACTER.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.78 SUBSTRING function (ODBC compatible)

Syntax

```
SUBSTRING ( char_expression, start_position [, length ] )
```

Description

The scalar function SUBSTRING returns the substring of the character string corresponding to the first argument starting at *start_position* and length characters long. If the third argument length is not specified, the substring starting at *start_position* up to the end of *char_expression* is returned. SUBSTRING is identical to SUBSTR but provides ODBC-compatible syntax.

Example

```
SELECT name, '(', SUBSTRING (phone, 1, 3) , ')',
               SUBSTRING (phone, 4, 3), '-',
               SUBSTRING (phone, 7, 4)
    FROM customer ;
```

Notes

- The first argument must be of type CHARACTER.
- The second argument must be of type INTEGER.
- The third argument, if specified, must be of type INTEGER.
- The values for specifying position in the character string start from 1: The very first character in a string is at position 1, the second character is at position 2 and so on.
- The result is of type CHARACTER.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.79 SUFFIX function (extension)

Syntax

```
SUFFIX(char_expression, start_position, char_expression)
```

Description

The scalar function SUFFIX returns the substring of a character string starting after the position specified by *start_position* and the second *char_expression*, to the end of the string.

Arguments

char_expression

An expression that evaluates to a character string, typically a character-string literal or column name. If the expression evaluates to null, SUFFIX returns null.

start_position

An expression that evaluates to an integer value. SUFFIX searches the string specified in the first argument starting at that position. A value of 1 indicates the first character of the string.

char_expression

An expression that evaluates to a single character. SUFFIX returns the substring that begins with that character. If SUFFIX does not find the character after *start_position*, it returns null. If the expression evaluates to more than one character, SUFFIX ignores all but the first character.

Example

```
SELECT C1, C2, SUFFIX(C1, 6, '.') FROM T1;
C1           C2  SUFFIX(C1,6,.
--           --  -------------
test.pref    .
pref.test    s
2 records selected
SELECT C1, C2, SUFFIX(C1, 1, C2) FROM T1;
C1           C2  SUFFIX(C1,1,C
--           --  -------------
test.pref    .   pref
pref.test    s   t
2 records selected
SELECT C1, C2, SUFFIX(C1, 6, '.') FROM T1;
C1           C2  SUFFIX(C1,6,.
--           --  -------------
test.pref    .
pref.test    s
2 records selected
```

1.10.2.80 SUSER_NAME function (extension)

Syntax

```
SUSER_NAME ( [user_id] )
```

Description

The scalar function SUSER_NAME returns the user login name for the *user_id* specified in the input argument. If no *user_id* is specified, SUSER_NAME returns the name of the current user.

This function provides compatibility with the Sybase SQL Server function *suser_name*. It is identical to the USER_NAME function.

Example

```
 select suser_name() from systpe.syscalctable;
SUSER_NAME
----------
searle
1 record selected
 select suser_name(104) from systpe.syscalctable;
SUSER_NAME(104)
---------------
dbp
1 record selected
  select id, tbl, owner from systpe.systables
   1  where owner = suser_name();
          ID TBL                                  OWNER
          -- ---                                  -----
          41 test                                 searle
          42 t2                                   searle
          43 t1                                   searle

3 records selected
```

1.10.2.81 SYSDATE function (extension)

Syntax

```
SYSDATE [ ( ) ]
```

Description

SYSDATE returns the current date as a DATE value. This function takes no arguments, and the trailing parentheses are optional.

SQL statements can refer to SYSDATE anywhere they can refer to a DATE expression. For example,

```
INSERT INTO objects (object_owner, object_id, create_date)
           VALUES (USER, 1001, SYSDATE) ;
```

1.10.2.82 SYSTIME function (extension)

Syntax

```
SYSTIME [ ( ) ]
```

Description

SYSTIME returns the current time as a TIME value. This function takes no arguments, and the trailing parentheses are optional.

SQL statements can refer to SYSTIME anywhere they can refer to a TIME expression. For example,

```
INSERT INTO objects (object_owner, object_id, create_time)
           VALUES (USER, 1001, SYSTIME) ;
```

1.10.2.83 SYSTIMESTAMP function (extension)

Syntax

```
SYSTIMESTAMP [ ( ) ]
```

Description

SYSTIMESTAMP returns the current date and time as a TIMESTAMP value. This function takes no arguments, and the trailing parentheses are optional.

The following SQL example shows the different formats for SYSDATE, SYSTIME, and SYSTIMESTAMP:

```
SELECT SYSDATE FROM test;
SYSDATE
-------
09/13/1994
1 record selected
SELECT SYSTIME FROM test;
SYSTIME
-------
14:44:07:000
1 record selected
SELECT SYSTIMESTAMP FROM test;
SYSTIMESTAMP
------------
1994-09-13 14:44:15:000
1 record selected
```

1.10.2.84 TAN function (ODBC compatible)

Syntax

```
TAN ( expression )
```

Description

The scalar function TAN returns the tangent of expression.

Example

```
select tan(45 * pi()/180) 'Tangent of 45 degrees' from syscal-
ctable;
TANGENT OF 45 DE
----------------
1.000000000000000
1 record selected
```

Notes

TAN takes an angle (expression) and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

- expression specifies an angle in radians
- expression must evaluate to an approximate numeric data type.

To convert degrees to radians, multiply degrees by Pi/180. To convert radians to degrees, multiply radians by 180/Pi.

1.10.2.85 TIMESTAMPADD function (ODBC compatible)

Syntax

```
TIMESTAMPADD(interval, integer_exp, date_time_exp)
interval::
          SQL_TSI_FRAC_SECOND
     |    SQL_TSI_SECOND
     |    SQL_TSI_MINUTE
     |    SQL_TSI_HOUR
     |    SQL_TSI_DAY
     |    SQL_TSI_WEEK
     |    SQL_TSI_MONTH
     |    SQL_TSI_QUARTER
     |    SQL_TSI_YEAR
```

Description

Returns the timestamp calculated by adding *integer_exp* intervals of type interval to *timestamp_exp*.

Arguments

interval

Keywords that specify the interval to add to *timestamp_exp*. The SQL_TSI_FRAC_SECOND keyword specifies fractional seconds as billionths of a second.

integer_exp

The number of interval values to add to *timestamp_exp*. *integer_exp* can be any expression that evaluates to an integer data type.

date_time_exp

A date-time expression from which TIMESTAMPADD calculates the return value. If *date_time_exp* is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of *timestamp_exp* is set to 0 before calculating the resulting timestamp.

Example

The following example displays the current system time and uses the TIMESTAMPADD scalar function to add 8 hours to it.

```
> select systime, timestampadd(sql_tsi_hour, 8, systime) from
syscalctable;
15:03:57:000    06/08/1999 23:03:57:000
------------    ----------------------
15:03:57:000,   1999-06-08 23:03:57:000
```

1.10.2.86 TIMESTAMPDIFF function (ODBC compatible)

Syntax

```
TIMESTAMPDIFF(interval, date_time_exp1, date_time_exp2)
interval::
          SQL_TSI_FRAC_SECOND
     |    SQL_TSI_SECOND
     |    SQL_TSI_MINUTE
     |    SQL_TSI_HOUR
     |    SQL_TSI_DAY
     |    SQL_TSI_WEEK
     |    SQL_TSI_MONTH
     |    SQL_TSI_QUARTER
     |    SQL_TSI_YEAR
```

Description

Returns an integer representing the number of intervals by which *date_time_exp2* is greater than *date_time_exp1*.

Arguments

interval

Keywords that specify the interval in which to express the difference between the two date-time arguments. The SQL_TSI_FRAC_SECOND keyword specifies fractional seconds as billionths of a second.

date_time_exp1

A date-time expression which TIMESTAMPADD subtracts from *date_time_exp2*.

date_time_exp1

A date-time expression from which TIMESTAMPADD subtracts *date_time_exp1*.

Example

The following example displays difference in seconds between the current system time and one day later.

```
> select timestampdiff(sql_tsi_second, sysdate, sysdate + 1)
from syscalctable;
86400
-----
86400
```

Notes

If either date-time expression is a time value and interval specifies days, weeks, months, quarters, or years, the date portion of that expression is set to the current date before calculating the difference between the expressions.

If either date-time expression is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of that expression is set to 0 before calculating the difference between the expressions.

1.10.2.87 TO_CHAR function (extension)

Syntax

```
TO_CHAR ( expression [ , format_string ] )
```

Description

The scalar function TO_CHAR converts the given expression to character form and returns the result. The primary use for TO_CHAR is to format the output of date-time expressions through the *format_string* argument.

Arguments

expression

Specifies the expression to be converted to character form. To use the *format_string* argument, *expression* must evaluate to a date or time value.

format_string

A date-time format string that specifies the format of the output. See *Date Format Strings* and *Time Format Strings* on page 1-41 for details on format strings.

SQL ignores the format string if the expression argument does not evaluate to a date or time.

Example

```
SELECT C1 FROM T2;
C1
--
09/29/1952
1 record selected
SELECT TO_CHAR(C1, 'Day, Month ddth'),
         TO_CHAR(C2, 'HH12 a.m.') FROM T2;
TO_CHAR(C1,DAY, MONTH DDTH)  TO_CHAR(C2,HH12 A.M.)
---------------------------  --------------------
Monday   , September 29th    02 p.m.
1 record selected
```

Notes

- The first argument to the function can be of any type.
- The second argument, if specified, must be of type CHARACTER.
- The result is of type CHARACTER.
- The format argument can be used only when the type of the first argument is DATE.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.88 TO_DATE function (extension)

Syntax

```
TO_DATE ( date_lit )
```

Description

The scalar function TO_DATE converts the given date literal to a date value.

Example

```
SELECT *
     FROM  orders
     WHERE order_date <= TO_DATE ('12/31/1991') ;
```

Notes

- The result is of type DATE.

- Supply the date literal in any valid format. See *Date-Time Literals* on page 1-37 for valid formats.

1.10.2.89 TO_NUMBER function (extension)

Syntax

```
TO_NUMBER ( char_expression )
```

Description

The scalar function TO_NUMBER converts the given character expression to a number value.

Example

```
SELECT *
     FROM  customer
     WHERE TO_NUMBER (SUBSTR (phone, 1, 3)) = 603 ;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type NUMERIC.
- If any of the argument expressions evaluates to null, the result is null.

1.10.2.90 TO_TIME function (extension)

Syntax

```
TO_TIME ( time_lit )
```

Description

The scalar function TO_TIME converts the given time literal to a time value.

Example

```
SELECT *
     FROM  orders
     WHERE order_date < TO_DATE ('05/15/1991')
           AND order_time < TO_TIME ('12:00:00') ;
```

Notes

- The result is of type TIME.
- Supply the time literal in any valid format. See *Date-Time Literals* on page 1-37 for valid formats.

1.10.2.91 TO_TIMESTAMP function (extension)

Syntax

```
TO_TIMESTAMP ( timestamp_lit )
```

Description

The scalar function TO_TIMESTAMP converts the given timestamp literal to a timestamp value.

Example

```
SELECT * FROM DTEST
WHERE C3 = TO_TIMESTAMP('4/18/95 10:41:19')
```

Notes

- The result is of type TIME.
- Supply the timestamp literal in any valid format. See *Timestamp Literals* on page 1-40 for valid formats.

1.10.2.92 TRANSLATE function (extension)

Syntax

```
TRANSLATE ( char_expression, from_set, to_set )
```

Description

The scalar function TRANSLATE translates each character in *char_expression* that is in *from_set* to the corresponding character in *to_set*. The translated character string is returned as the result. This function is similar to the Oracle TRANSLATE function.

Example

This example substitutes underscores for spaces in customer names.

```
SELECT TRANSLATE (customer_name, ' ', '_')
      "TRANSLATE Example" from customers;
TRANSLATE EXAMPLE
-----------------
Sports_Cars_Inc._____________________________
Mighty_Bulldozer_Inc._____________________________
Ship_Shapers_Inc._____________________________
Tower_Construction_Inc._____________________________
Chemical_Construction_Inc._____________________________
Aerospace_Enterprises_Inc._____________________________
Medical_Enterprises_Inc._____________________________
Rail_Builders_Inc._____________________________
Luxury_Cars_Inc._____________________________
```

```
Office_Furniture_Inc._____________________________
10 records selected
```

Notes

- *char_expression*, *from_set*, and *to_set* can be any character expression.
- For each character in *char_expression*, TRANSLATE checks for the same character in *from_set*:
- If it is in *from_set*, TRANSLATE translates it to the corresponding character in *to_set* (if the character is the nth character in *from_set*, the nth character in *to_set*).
- If the character is not in *from_set* TRANSLATE does not change it.
- If *from_set* is longer than *to_set*, TRANSLATE does not change trailing characters in *from_set* that do not have a corresponding character in *to_set*.
- If either *from_set* or *to_set* is null, TRANSLATE does nothing.

1.10.2.93 UCASE function (ODBC compatible)

Syntax

```
UCASE ( char_expression )
```

Description

The scalar function UCASE returns the result of the argument character expression after converting all the characters to upper case. UCASE is identical to UPPER, but provides ODBC-compatible syntax.

Example

```
SELECT *
     FROM  customer
     WHERE UCASE (name) = 'SMITH' ;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

1.10.2.94 UID function (extension)

UID returns an integer identifier for the user of the current transaction, as determined by the host operating system.

The value contained in this register is of INTEGER type. The host representation is of long integer type. SQL statements can refer to UID anywhere they can refer to an integer expression. For example,

```
INSERT INTO objects (owner_id, object_id)
```

```
                    VALUES (UID, 1001) ;

SELECT *
    FROM objects
    WHERE owner_id = UID ;
```

1.10.2.95 UPPER function (SQL-92 compatible)

Syntax

```
UPPER ( char_expression )
```

Description

The scalar function UPPER returns the result of the argument character expression after converting all the characters to upper case.

Example

```
SELECT *
     FROM  customer
     WHERE UPPER (name) = 'SMITH' ;
```

Notes

- The argument to the function must be of type CHARACTER.
- The result is of type CHARACTER.
- If the argument expression evaluates to null, the result is null.

1.10.2.96 USER function (ODBC compatible)

Syntax

```
USER [ ( ) ]
```

Description

USER returns a character-string identifier for the database user, as specified in the current connection. If the current connection did not specify a user, USER returns the login name as determined by the host operating system. This function takes no arguments, and the trailing parentheses are optional.

SQL statements can refer to USER anywhere they can refer to a character string expression.

Example

The following interactive SQL example shows connecting to a database as the user *fred*. Queries on two system tables illustrate the USER scalar function and retrieve the names of any tables owned by the user *fred*:

```
% isql -u fred tstdb
```

```
                Dharma/isql Version 9.00.00
        Dharma Systems Inc           (C) 1988-2004.
        Dharma Computers Pvt Ltd     (C) 1988-2004.
ISQL> select user from systpe.syscalctable;
FRED
----
fred
1 record selected
ISQL> select tbl, owner from systpe.systables where owner =
user();
TBL                              OWNER
---                              -----
flab                              fred
1 record selected
```

1.10.2.97 USER_NAME function (extension)

Syntax

```
USER_NAME ( [user_id] )
```

Description

The scalar function USER_NAME returns the user login name for the *user_id* specified in the input argument. If no *user_id* is specified, USER_NAME returns the name of the current user.

The scalar function USER_NAME is identical to SUSER_NAME.

1.10.2.98 WEEK function (ODBC compatible)

Syntax

```
WEEK ( time_expression )
```

Description

The scalar function WEEK returns the week of the year as a short integer value in the range of 1 - 53.

Example

```
SELECT *
     FROM  orders
     WHERE WEEK (order_date) = 5 ;
```

Notes

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

1.10.2.99 YEAR function (ODBC compatible)

Syntax

```
YEAR ( date_expression )
```

Description

The scalar function YEAR returns the year as a short integer value in the range of 0 - 9999.

Example

```
SELECT *
     FROM  orders
     WHERE YEAR (order_date) = 1992 ;
```

Notes

- The argument to the function must be of type DATE.
- The argument must be specified in the format MM/DD/YYYY.
- The result is of type SHORT.
- If the argument expression evaluates to null, the result is null.

Chapter 2

SQL Statements

2.1 INTRODUCTION

This chapter provides detailed reference material on each SQL statement.

2.2 CREATE INDEX

Description

Creates an index on the specified table using the specified columns of the table. An index improves the performance of SQL operations whose predicates are based on the indexed column. However, an index slows performance of INSERT, DELETE and UPDATE operations.

Syntax

```
CREATE [ UNIQUE ] INDEX index_name
       ON table_name
       ({column_name [ASC | DESC]} [, ...])
       [ PCTFREE number]
       [ STORAGE_ATTRIBUTES 'attributes' ]
       [ TYPE 'ix_type' ]
       ;
```

Arguments

UNIQUE
A UNIQUE index will not allow the table to contain any rows with duplicate column values for the set of columns specified for that index.

index_name
The name of the index has to be unique within the local database.

table_name
The name of the table on which the index is being built.

column_name [, ...]
The columns on which searches and retrievals will be ordered. These columns are called the index key. When more than one column is specified in the CREATE INDEX statement a concatenated index is created.

ASC | DESC
The index can be ordered as either ascending (ASC) or descending (DESC) on each column of the concatenated index. The default is ASC.

PCTFREE number
Specifies the desired percentage of free space for a index. The PCTFREE clause indicates to the storage system how much of the space allocated to an index should be left free to accommodate growth. However, the actual behavior of the PCTFREE clause depends entirely on the underlying storage system. The SQL engine passes the PCTFREE value to the storage system, which may ignore it or interpret it. If the CREATE statement does not include a PCTFREE clause, the default is 20. See the documentation for your storage system for details.

STORAGE_ATTRIBUTES 'attributes'
A quoted string that specifies index attributes that are specific to a particular storage system. The SQL engine passes this string to the storage system, and its effects are defined by the storage manager. See the documentation for your storage system for details.

TYPE 'ix_type'
A single-character that specifies the type of index. The valid values for the TYPE argument and their meanings are specific to the underlying storage system. See the documentation for your storage system for details.

Example

```
CREATE UNIQUE INDEX custindex ON customer (cust_no) ;
```

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege
- Ownership of the index.
- INDEX privilege on the table.

SQL Compliance	ODBC Core SQL grammar. Extensions: PCTFREE, STORAGE_ATTRIBUTES, and TYPE
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE TABLE, DROP INDEX, ALTER INDEX

2.3 CREATE SYNONYM

Description

Creates a synonym for the table, view or synonym specified. A synonym is an alias that SQL statements can use instead of the name specified when the table, view, or synonym was created.

Syntax

```
CREATE [PUBLIC] SYNONYM synonym
        FOR [owner_name.] { table_name | view_name | synonym } ;
```

Arguments

PUBLIC
Specifies that the synonym will be public: all users can refer to the name without qualifying it. By default, the synonym is private: other users must qualify the synonym by preceding it with the user name of the user who created it.

Users must have the DBA privilege to create public synonyms.

SYNONYM synonym
Name for the synonym.

FOR [owner_name.] { table_name | view_name | synonym }
Table, view, or synonym for which SQL creates the new synonym.

Example

```
CREATE SYNONYM customer FOR smith.customer ;
CREATE PUBLIC SYNONYM public_suppliers FOR smith.suppliers ;
```

Authorization

Users executing CREATE SYNONYM must have the DBA privilege or RESOURCE privilege. Users executing CREATE PUBLIC SYNONYM statement must have the DBA privilege.

SQL Compliance	Extension
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	DROP SYNONYM

2.4 CREATE TABLE

Description

Creates a table definition. A table definition consists of a list of column definitions that make up a table row. SQL provides two forms of the CREATE TABLE statement. The first form explicitly specifies column definitions. The second form, with the AS *query_expression* clause, implicitly defines the columns using the columns in the query expression.

Syntax

```
CREATE TABLE [ owner_name. ] table_name
( column_definition [ , { column_definition | table_constraint
} ] ... )
  [ TABLE SPACE table_space_name ]
  [ PCTFREE number ]
  [ STORAGE_MANAGER ‘sto-mgr-id’ ]
  [ STORAGE_ATTRIBUTES ‘attributes’ ]
  ;
CREATE TABLE [ owner_name. ] table_name
  [ ( column_name [NULL | NOT NULL], ...) ]
  [ TABLE SPACE table_space_name ]
  [ PCTFREE number ]
  [ STORAGE_MANAGER ‘sto-mgr-id’ ]
  [ STORAGE_ATTRIBUTES ‘attributes’ ]
  AS query_expression
  ;
column_definition ::
  column_name data_type
  [ COLLATE collation_name ]
  [ DEFAULT { literal | USER | NULL | UID
  | SYSDATE | SYSTIME | SYSTIMESTAMP } ]
  [ column_constraint [ column_constraint ... ] ]
```

Arguments

owner_name
Specifies the owner of the table. If the name is different from the user name of the user executing the statement, then the user must have DBA privileges.

table_name
Names the table definition. SQL defines the table in the database named in the last CONNECT statement.

column_name data_type
Names a column and associates a data type with it. The column names specified must be different than other column names in the table definition. The *data_type* must be one of the supported data types described in section "1.4 Data Types" on page 1-4.

[COLLATE collation_name]
If *data_type* specifies a character column, the column definition can include an optional COLLATE clause. The COLLATE clause specifies a collation sequence supported by the underlying storage system. (See "1.4.1.1 Specifying the Character Set for Character Data Types" on page 1-6 for notes on character sets and collations, including the CASE_INSENSITIVE collation sequence supported for the default character set. See the documentation for your underlying storage system for details on any supported collations.)

DEFAULT
Specifies an explicit default value for a column. The column takes on the value if an INSERT statement does not include a value for the column. If a column definition omits the DEFAULT clause, the default value is NULL.

The DEFAULT clause accepts the following arguments:

literal	An integer, numeric or string constant.
USER	The name of the user issuing the INSERT or UPDATE statement on the table. Valid only for columns defined with character data types.
NULL	A null value.
UID	The user id of the user executing the INSERT or UPDATE statement on the table.
SYSDATE	The current date. Valid only for columns defined with DATE data types.
SYSTIME	The current time. Valid only for columns defined with TIME data types.
SYSTIMESTAMP	The current date and time. Valid only for columns defined with TIMESTAMP data types.

column_constraint
Specifies a constraint that applies while inserting or updating a value in the associated column. For more information, see *Column Constraints* on page 2-7.

table_constraint
Specifies a constraint that applies while inserting or updating a row in the table. For more information, see *Table Constraints* on page 2-10.

TABLE SPACE table_space_name
Specifies the name of the table space where data stored in the table will reside. Table spaces provide a way to partition tables among different storage areas. In some storage systems, for instance, table spaces correspond to separate data files among which data in tables can be distributed. This arrangement can improve performance by distributing data in a table on different disk drives.

Different storage systems implement the concept of storage areas in different ways, if at all. So the actual behavior of the TABLE SPACE clause depends on the underlying storage system. See the documentation for your storage system for more details.

PCTFREE number
Specifies the desired percentage of free space for a table. The PCTFREE clause indicates to the storage system how much of the space allocated to a table should be left free to accommodate growth.

However, the actual behavior of the PCTFREE clause depends entirely on the underlying storage system. The SQL engine passes the PCTFREE value to the storage system, which may ignore it or interpret it. If the CREATE statement does not include a PCTFREE clause, the default is 20. See the documentation for your storage system for details.

STORAGE_MANAGER 'sto-mgr-id'
A quoted string that identifies the storage system. The SQL engine uses the string to identify which storage system will create the table. If the CREATE TABLE statement omits the STORAGE_MANAGER clause, the SQL engine uses the string 'default'. What constitutes a valid name, and how the table is mapped to a specific storage system is defined by the implementation.

STORAGE_ATTRIBUTES 'attributes'
A quoted string that specifies table attributes that are specific to a particular storage system. The SQL engine passes this string to the storage system, and its effects are defined by the storage manager. See the documentation for your storage system for details.

AS query_expression
Specifies a query expression to use for the data types and contents of the columns for the table. The types and lengths of the columns of the query expression result become the types and lengths of the respective columns in the table created. The rows in the resultant set of the query expression are inserted into the table after creating the table. In this form of the CREATE TABLE statement, column names are optional.

If omitted, the names for the table columns are also derived from the query expression. For more information, see "1.5 Query Expressions" on page 1-14.

Examples

In the following example, the user issuing the CREATE TABLE statement must have REFERENCES privilege on the column *itemno* of the table *john.item*.

```
CREATE TABLE supplier_item (
  supp_no     INTEGER NOT NULL PRIMARY KEY,
  item_no     INTEGER NOT NULL REFERENCES john.item (itemno),
  qty         INTEGER
) ;
```

The following CREATE TABLE statement explicitly specifies a table owner, systpe:

```
CREATE TABLE systpe.account (
          account  integer,
```

```
        balance  money (12),
        info     char (84)
    ) ;
```

The following example shows the AS *query_expression* form of CREATE TABLE to create and load a table with a subset of the data in the customer table:

```
CREATE TABLE systpe.dealer (name, street, city, state)
        AS
        SELECT name, street, city, state
        FROM customer
        WHERE customer.state IN ('CA','NY', 'TX') ;
```

The following example includes a NOT NULL column constraint and DEFAULT clauses for column definitions:

```
    CREATE TABLE emp (
        empno  integer NOT NULL,
        deptno  integer DEFAULT 10,
        join_date date DEFAULT NULL
    ) ;
```

Authorization

The user executing this statement must have either DBA or RESOURCE privilege. If the CREATE TABLE statement specifies a foreign key that references a table owned by a different user, the user must have the REFERENCES privilege on the corresponding columns of the referenced table.

The AS *query_expression* form of CREATE TABLE requires the user to have select privilege on all the tables and views named in the query expression.

SQL Compliance	SQL-92, ODBC Minimum SQL grammar. Extensions: TABLE SPACE, PCTFREE, STORAGE_MANAGER, and AS *query_expression*
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	DROP TABLE, Query Expressions

2.4.1 Column Constraints

Description

Specifies a constraint for a column that restricts the values that the column can store. INSERT, UPDATE, or DELETE statements that violate the constraint fail. SQL returns a Constraint violation error with SQLCODE of -20116.

Column constraints are similar to table constraints but their definitions are associated with a single column.

Syntax

```
column_constraint ::
  NOT NULL [ PRIMARY KEY | UNIQUE ]
| REFERENCES [ owner_name. ] table_name [ ( column_name ) ]
| CHECK ( search_condition )
```

Arguments

NOT NULL
Restricts values in the column to values that are not null.

NOT NULL PRIMARY KEY
Defines the column as the primary key for the table. There can be only one primary key for a table. A column with the NOT NULL PRIMARY KEY constraint cannot contain null or duplicate values. Other tables can name primary keys as foreign keys in their REFERENCES clauses.

Other tables can name primary keys in their REFERENCES clauses. If they do, SQL restricts operations on the table containing the primary key:

- DROP TABLE statements that delete the table fail
- DELETE and UPDATE statements that modify values in the column that match a foreign key's value also fail

The following example shows the creation of a primary key column on the table supplier.

```
CREATE TABLE supplier (
          supp_no     INTEGER NOT NULL PRIMARY KEY,
          name        CHAR (30),
          status      SMALLINT,
          city        CHAR (20)
     ) ;
```

NOT NULL UNIQUE
Defines the column as a unique key that cannot contain null or duplicate values. Columns with NOT NULL UNIQUE constraints defined for them are also called candidate keys.

Other tables can name unique keys in their REFERENCES clauses. If they do, SQL restricts operations on the table containing the unique key:

- DROP TABLE statements that delete the table fail
- DELETE and UPDATE statements that modify values in the column that match a foreign key's value also fail

The following example creates a NOT NULL UNIQUE constraint to define the column *ss_no* as a unique key for the table *employee*:

```
CREATE TABLE employee (
          empno        INTEGER NOT NULL PRIMARY KEY,
```

```
        ss_no        INTEGER NOT NULL UNIQUE,
        ename        CHAR (19),
        sal          NUMERIC (10, 2),
        deptno       INTEGER NOT NULL
    ) ;
```

REFERENCES table_name [(column_name)]
Defines the column as a foreign key and specifies a matching primary or unique key in another table. The REFERENCES clause names the matching primary or unique key.

A foreign key and its matching primary or unique key specify a referential constraint: A value stored in the foreign key must either be null or be equal to some value in the matching unique or primary key.

You can omit the *column_name* argument if the table specified in the REFERENCES clause has a primary key and you want the primary key to be the matching key for the constraint.

The following example defines *order_item.orditem_order_no* as a foreign key that references the primary key *orders.order_no*.

```
CREATE TABLE orders (
  order_no INTEGER NOT NULL PRIMARY KEY,
  order_date DATE
) ;
CREATE TABLE order_item (
  orditem_order_no INTEGER REFERENCES orders ( order_no ),
  orditem_quantity INTEGER
) ;
```

Note that the second CREATE TABLE statement in the previous example could have omitted the column name *order_no* in the REFERENCES clause, since it refers to the primary key of table orders.

CHECK (search_condition)
Specifies a column-level check constraint. SQL restricts the form of the search condition. The search condition must not:

- Refer to any column other than the one with which it is defined
- Contain aggregate functions, subqueries, or parameter references

The following example creates a check constraint:

```
CREATE TABLE supplier (
    supp_no     INTEGER NOT NULL,
    name        CHAR (30),
    status      SMALLINT,
    city        CHAR (20) CHECK (supplier.city <> 'MOSCOW')
) ;
```

2.4.2 Table Constraints

Description

Specifies a constraint for a table that restricts the values that the table can store. INSERT, UPDATE, or DELETE statements that violate the constraint fail. SQL returns a Constraint violation error.

Table constraints have syntax and behavior similar to column constraints. Note the following differences:

- The syntax for table constraints is separated from column definitions by commas.
- Table constraints must follow the definition of columns they refer to.
- Table constraint definitions can include more than one column and SQL evaluates the constraint based on the combination of values stored in all the columns.

Syntax

```
table_constraint ::
 PRIMARY KEY ( column [, ... ] )
 | UNIQUE ( column [, ... ] )
 | FOREIGN KEY ( column [, ... ] )
   REFERENCES [ owner_name. ] table_name [ ( column [, ... ] ) ]
 | CHECK ( search_condition )
```

Arguments

PRIMARY KEY (column [, ...])
Defines the column list as the primary key for the table. There can be at most one primary key for a table.

All the columns that make up a table-level primary key must be defined as NOT NULL, or the CREATE TABLE statement fails. The combination of values in the columns that make up the primary key must be unique for each row in the table.

Other tables can name primary keys in their REFERENCES clauses. If they do, SQL restricts operations on the table containing the primary key:

- DROP TABLE statements that delete the table fail
- DELETE and UPDATE statements that modify values in the combination of columns that match a foreign key's value also fail

The following example shows creation of a table-level primary key. Note that its definition is separated from the column definitions by a comma:

```
CREATE TABLE supplier_item (
     supp_no   INTEGER NOT NULL,
     item_no   INTEGER NOT NULL,
     qty       INTEGER NOT NULL DEFAULT 0,
          PRIMARY KEY (supp_no, item_no)
) ;
```

UNIQUE (column [, ...])
Defines the column list as a unique, or candidate, key for the table. Unique key table-level constraints have the same rules as primary key table-level constraints, except that you can specify more than one UNIQUE table-level constraint in a table definition.

The following example shows creation of a table with two UNIQUE table-level constraints:

```
CREATE TABLE order_item (
     order_no    INTEGER NOT NULL,
     item_no     INTEGER NOT NULL,
     qty         INTEGER NOT NULL,
     price       MONEY NOT NULL,
          UNIQUE (order_no, item_no),
          UNIQUE (qty, price)
) ;
```

FOREIGN KEY ... REFERENCES
Defines the first column list as a foreign key and, in the REFERENCES clause, specifies a matching primary or unique key in another table.

A foreign key and its matching primary or unique key specify a referential constraint: The combination of values stored in the columns that make up a foreign key must either:

- Have at least one of the column values be null
- Be equal to some corresponding combination of values in the matching unique or primary key

You can omit the column list in the REFERENCES clause if the table specified in the REFERENCES clause has a primary key and you want the primary key to be the matching key for the constraint.

The following example defines the combination of columns student_courses.teacher and student_courses.course_title as a foreign key that references the primary key of the table *courses*. Note that the REFERENCES clause does not specify column names because the foreign key refers to the primary key of the courses table.

```
CREATE TABLE courses (
     teacher        CHAR (20) NOT NULL,
     course_title   CHAR (30) NOT NULL,
     PRIMARY KEY (teacher, course_title)
) ;
CREATE TABLE student_courses (
     student_id     INTEGER,
     teacher        CHAR (20),
     course_title   CHAR (30),
     FOREIGN KEY (teacher, course_title) REFERENCES courses
```

```
) ;
```

SQL evaluates the referential constraint to see if it satisfies the following search condition:

```
(student_courses.teacher IS NULL
  OR student_courses.course_title IS NULL)
OR
EXISTS (SELECT * FROM student_courses WHERE
     (student_courses.teacher = courses.teacher AND
      student_courses.course_title = courses.course_title)
        )
```

INSERT, UPDATE or DELETE statements that cause the search condition to be false violate the constraint, fail, and generate an error.

CHECK (search_condition)
Specifies a table-level check constraint. The syntax for table-level and column level check constraints is identical. Table-level check constraints must be separated by commas from surrounding column definitions.

SQL restricts the form of the search condition. The search condition must not:

- Refer to any column other than columns that precede it in the table definition
- Contain aggregate functions, subqueries, or parameter references

The following example creates a table with two column-level check constraints and one table-level check constraint:

```
CREATE TABLE supplier (
   supp_no   INTEGER NOT NULL,
   name      CHAR (30),
   status    SMALLINT CHECK (
         supplier.status BETWEEN 1 AND 100 ),
   city      CHAR (20) CHECK (
         supplier.city IN ('NEW YORK', 'BOSTON', 'CHICAGO')),
   CHECK (supplier.city <> 'CHICAGO' OR supplier.status = 20)
) ;
```

2.5 CREATE VIEW

Description

Creates a view with the specified name on existing tables and/or views.

Syntax

```
CREATE VIEW [ owner_name. ] view_name
       [ ( column_name, column_name,... ) ]
       AS [ ( ] query_expression [ ) ]
       [ WITH CHECK OPTION ] ;
```

Notes

- The owner_name is made the owner of the created view.
- The column names specified for the view are optional and provide an alias for the columns selected by the query specification. If the column names are not specified then the view will be created with the same column names as the tables and/or views it is based on.
- A view is deletable if deleting rows from that view is allowed. For a view to be deletable, the view definition has to satisfy the following conditions:
  - The first FROM clause contains only one table reference or one view reference.
  - There are no aggregate functions, DISTINCT clause, GROUP BY or HAVING clause in the view definition.
  - If the first FROM clause contains a view reference, then the view referred to is deletable.
- A view is updatable if updating rows from that view is allowed. For a view to be updatable, the view has to satisfy the following conditions:
  - The view is deletable (That is, it satisfies all the conditions specified above for deletability).
  - All the select expressions in the first SELECT clause of the view definition are simple column references.
  - If the first FROM clause contains a view reference, then the view referred to is updatable.
- A view is insertable if inserting rows into that view is allowed. For a view to be insertable, the view has to satisfy the following conditions:
  - The view is updatable (That is, it satisfies all the conditions specified above for updatability).
  - If the first FROM clause contains a table reference, then all NOT NULL columns of the table are selected in the first SELECT clause of the view definition.

  - If the first FROM clause contains a view reference, then the view referred to is insertable.

- The WITH CHECK OPTION clause can be specified only if the view is updatable.
- If WITH CHECK OPTION clause is specified when defining a view, then during any update or insert of a row on this view, it is checked that the updated/inserted row satisfies the view definition (That is, the row is selectable using the view).

Examples

```
CREATE VIEW ne_customers AS
     SELECT cust_no, name, street, city, state, zip
     FROM customer
     WHERE state IN ('NH', 'MA', 'NY', 'VT')
     WITH CHECK OPTION ;
CREATE VIEW order_count (cust_number, norders) AS
     SELECT cust_no, COUNT(*)
     FROM orders
     GROUP BY cust_no;
```

Authorization

The user executing this statement must have the following privileges:

- DBA or RESOURCE privilege.
- SELECT privilege on all the tables/views referred to in the view definition.

If owner_name is specified and is different from the name of the user executing the statement, then the user must have DBA privilege.

SQL Compliance	SQL-92, ODBC Core SQL grammar
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	Query Expressions, DROP VIEW

2.6 DELETE

Description

Deletes zero, one or more rows from the specified table that satisfy the search condition specified in the WHERE clause. If the optional WHERE clause is not specified, then the DELETE statement deletes all rows of the specified table.

Syntax

```
DELETE FROM [owner_name.] { table_name | view_name }
          [ WHERE search_condition ];
```

Notes

- If the table has primary/candidate keys, and if there exists references from other tables to the rows to be deleted, the statement is rejected.
- Deletion of data in linked tables is supported. However, the usage of the query expression in the search condition of the DELETE statement is not supported. Deletes to local views with CHECK option return error as the feature is not supported.

Example

```
DELETE FROM customer
               WHERE customer_name = 'RALPH' ;
```

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege.
- Ownership of the table.
- DELETE permission on the table.

If the target is a view, then the DELETE privilege is required on the target base table referred to in the view definition.

SQL Compliance	SQL-92, ODBC Extended SQL grammar
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	Search Conditions

2.7 DROP INDEX

Description

Deletes an index on the specified table.

Syntax

```
DROP INDEX [index_owner_name.]index_name
          [ON [table_owner_name.]table_name]
```

Arguments

index_owner_name
If *index_owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

table_name
The *table_name* argument is optional. If specified, the *index_name* is verified to correspond to the table.

Example

```
DROP INDEX custindex ON customer ;
```

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege
- Ownership of the index

SQL Compliance	ODBC Core SQL grammar
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE INDEX, ALTER INDEX

2.8 DROP SYNONYM

Description

Drops the specified synonym.

Syntax

```
DROP [PUBLIC] SYNONYM [owner_name.]synonym ;
```

Arguments

PUBLIC
Specifies that the synonym was created with the PUBLIC argument.

SQL generates the *Base table not found* error if DROP SYNONYM specifies PUBLIC and the synonym was not a public synonym. Conversely, the same error message occurs if DROP SYNONYM does not specify public and the synonym was created with the PUBLIC argument.

To drop a public synonym, you must have the DBA privilege.

owner_name
If *owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

synonym
Name for the synonym.

Example

```
DROP SYNONYM customer ;
DROP PUBLIC SYNONYM public_suppliers ;
```

Authorization

Users executing DROP SYNONYM must have either the DBA privilege or be the owner of the synonym. Users executing DROP PUBLIC SYNONYM must have the DBA privilege.

SQL Compliance	Extension
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE SYNONYM

2.9 DROP TABLE

Description

Deletes the specified table.

Syntax

```
DROP TABLE  [owner_name.]table_name ;
```

Notes

- If *owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.
- When a table is dropped, the indexes on the table and the privileges associated with the table are dropped automatically.
- Views dependent on the dropped table are not automatically dropped, but become invalid.
- If the table is part of another table's referential constraint (if the table is named in another table's REFERENCES clause), the DROP TABLE statement fails. Use the ALTER TABLE statement to delete any referential constraints that refer to the table before issuing the DROP TABLE statement.

Example

```
DROP TABLE customer ;
```

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege.
- Ownership of the table.

SQL Compliance	SQL-92, ODBC Minimum SQL grammar
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE TABLE

2.10 DROP VIEW

Description

Deletes the view from the database.

Syntax

```
DROP VIEW [owner_name.]view_name ;
```

Notes

If *owner_name* is specified and is different from the name of the user executing the statement, then the user must have DBA privileges.

When a view is dropped, other views that are dependent on this view are not dropped. The dependent views become invalid.

Example

```
DROP VIEW newcustomers ;
```

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege
- Ownership of the view

SQL Compliance	SQL-92, ODBC Core SQL grammar
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE VIEW

2.11 GRANT

Description

Grants various privileges to the specified users for the database. There are different forms of the GRANT statement for various purposes:

- To grant database-wide privileges, either system administration (DBA) or general creation (RESOURCE)
- To grant privileges on the specified tables or view
- To grant the privilege to execute the specified stored procedure

Syntax

```
GRANT { RESOURCE, DBA }
    TO user_name [ , user_name ] ...  ;

GRANT { privilege [ , privilege ] ... | ALL [ PRIVILEGES ] }
    ON table_name
    TO  { user_name [ , user_name ] ... | PUBLIC }
    [WITH GRANT OPTION] ;

GRANT EXECUTE ON procedure_name
    TO  { user_name [ , user_name ] ... | PUBLIC } ;

privilege ::
    { SELECT | INSERT | DELETE | ALTER | INDEX
            | UPDATE [ (column, column, ... ) ]
            | REFERENCES [ (column, column, ... ) ] }
```

Arguments

DBA
Allows the specified users to create, access, modify, or delete any database object, and to grant other users any privileges.

RESOURCE
Allows the specified users to issue CREATE statements. The RESOURCE privilege does not allow users to issue DROP statements on database objects. Only the owner of the object and users with the DBA privilege can drop database objects.

SELECT
Allows the specified users to read data in the table or view.

INSERT
Allows the specified users to add new rows to the table or view.

DELETE
Allows the specified users to delete rows in the table or view

ALTER
Allows the specified users to modify the table or view

INDEX
Allows the specified users to create an index on the table or view.

UPDATE [(column, column, ...)]
Allows the specified users to modify existing rows in the table or view. If followed by a column list, the users can modify values only in the columns named.

REFERENCES [(column, column, ...)]
Allows the specified users to refer to the table from other tables' constraint definitions. If followed by a column list, constraint definitions can refer only to the columns named. For more detail on constraint definitions, see *Column Constraints* on page page 2-7.

ALL
Grants all privileges for the table or view.

ON table_name
The table or view for which SQL grants the specified privileges.

EXECUTE ON procedure_name
Allows execution of the specified stored procedure.

TO user_name [, user_name] ...
The list of users for which SQL grants the specified privileges.

TO PUBLIC
Grants the specified privileges to any user with access to the system.

WITH GRANT OPTION
Allows the specified users to grant their access rights or a subset of their rights to other users.

Example

```
GRANT ALTER ON cust_view TO dbuser1 ;
GRANT SELECT ON newcustomers TO dbuser2 ;
GRANT EXECUTE ON sample_proc TO searle;
```

Authorization

The user granting DBA or RESOURCE privileges must have the DBA privilege.

The user granting privileges on a table must have any of the following privileges:

- DBA privilege
- Ownership of the table
- All the specified privileges on the table, granted with the WITH GRANT OPTION clause

SQL Compliance SQL-92, ODBC Core SQL grammar. Extensions: ALTER, INDEX, RESOURCE, DBA privileges

Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	REVOKE

2.12 INSERT

Description

Inserts new rows into the specified table/view that will contain either the explicitly specified values or the values returned by the query expression.

Syntax

```
INSERT INTO [owner_name.] { table_name | view_name }
          [ (column_name, column_name, ...) ]
          { VALUES (value, value, ...) | query_expression };
```

Notes

- If the optional list of column names is specified, then only the values for those columns need be supplied. The rest of the columns of the inserted row will contain NULL values, provided the table definition allows NULL values and there is no DEFAULT clause for the columns. If a DEFAULT clause is specified for a column and the column name is not present in the optional column list, then the column takes the default value.
- If the optional list is not specified then all the column values have to be either explicitly specified or returned by the query expression. The order of the values should be the same as the order in which the columns have been declared in the declaration of the table/view.
- Explicit specification of the column values provides for insertion of only one row at a time. The query expression option allows for insertion of multiple rows at a time.
- If the table contains a foreign key, and there does not exist a corresponding primary key that matches the values of the foreign key in the record being inserted, the insert operation is rejected.
- You can use INSERT statements with query expressions to transfer rows from one remote table to another.

Examples

```
INSERT INTO customer (cust_no, name, street, city, state)
       VALUES
         (1001, 'RALPH', '#10 Columbia Street', 'New York',
'NY') ;


     INSERT INTO neworders (order_no, product, qty)
          SELECT order_no, product, qty
          FROM orders
          WHERE order_date = SYSDATE ;
```

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege.
- Ownership of the table.
- INSERT privilege on the table.

If a *query_expression* is specified, then the user must have any of the following privileges:

- DBA privilege.
- SELECT privilege on all the tables/views referred to in the *query_expression*.

SQL Compliance	SQL-92, ODBC Core SQL grammar
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	Query Expressions

2.13 RENAME

Description

Renames the specified table name, view name or synonym to the new name specified.

Syntax

```
RENAME [owner_name.] oldname TO [owner_name.] newname ;
```

Arguments

[owner_name.]
Optional owner-name qualifier for the name. If the owner name is not the same as that of the current user, the current user must have the DBA privilege.

If specified, the owner name must be the same for *oldname* and *newname*. In other words, you cannot change the owner of a table, view, or synonym with RENAME.

oldname
Current name of the table, view, or synonym.

newname
New name for the table, view, or synonym.

Example

```
RENAME sitem TO supplier_item ;
```

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege
- Ownership of the table/view/synonym.
- ALTER privilege on the table/view.

SQL Compliance	Extension
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	CREATE TABLE, CREATE VIEW, CREATE SYNONYM

2.14 REVOKE

Description

Revokes various privileges to the specified users for the database. There are three forms of the REVOKE statement:

- The first form revokes database-wide privileges, either system administration (DBA) or general creation (RESOURCE)
- The second form revokes various privileges on specific tables and views
- The third form revokes the privilege to execute the specified stored procedure

Syntax

```
REVOKE  { RESOURCE | DBA }
          FROM { user_name [ , user_name ] ... } ;

REVOKE [ GRANT OPTION FOR ]
         { privilege [ , privilege, ] ... | ALL [ PRIVILEGES ] }
          ON table_name
          FROM { user_name [ , user_name ] ... | PUBLIC } [
RESTRICT | CASCADE ] ;

REVOKE [ GRANT OPTION FOR ] EXECUTE ON procedure_name
          FROM { user_name [ , user_name ] ... | PUBLIC } [
RESTRICT | CASCADE ] ;

privilege ::
    { SELECT | INSERT | DELETE | ALTER | INDEX
            | UPDATE [ (column, column, ... ) ]
            | REFERENCES [ (column, column, ... ) ] }
```

Arguments

GRANT OPTION FOR
Revokes the grant option for the privilege from the specified users. The actual privilege itself is not revoked. If specified with RESTRICT, and the privilege was passed on to other users, the REVOKE statement fails and generates an error. Otherwise, GRANT OPTION FOR implicitly revokes any rights the user may have in turn given to other users.

{ privilege [, privilege,] ... | ALL [PRIVILEGES] }
List of privileges to be revoked. See the description in the GRANT statement on page page 20 for details on specific privileges. Revoking RESOURCE and DBA rights can only be done by the administrator or a user with DBA rights.

If a user has been granted access to a table by more than one user then all the users have to perform a revoke for the user to lose his access to the table.

Using the keyword ALL revokes all the rights granted on the table/view.

ON table_name
The table or view for which SQL revokes the specified privileges.

EXECUTE ON procedure_name
Revokes the right to execute the specified stored procedure.

FROM user_name [, user_name] ...
Revokes the specified rights on the table or view from the specified list of users.

FROM PUBLIC
Revokes the specified rights on the table or view from any user with access to the system.

RESTRICT | CASCADE
If the REVOKE statement specifies RESTRICT, SQL checks to see if the privilege being revoked was passed on to other users (possible only if the original privilege included the WITH GRANT OPTION clause). If so, the REVOKE statement fails and generates an error. If the privilege was not passed on, the REVOKE statement succeeds.

If the REVOKE statement specifies CASCADE, revoking the access right of a user also revokes the rights from all users who received the privilege as a result of that user giving the privilege to others.

If the REVOKE statement specifies neither RESTRICT nor CASCADE, the behavior is the same as for CASCADE.

Example

```
REVOKE INSERT ON customer FROM dbuser1 ;
REVOKE ALTER ON cust_view FROM dbuser2 ;
```

Authorization

The user revoking DBA or RESOURCE privileges must have the DBA privilege.

The user revoking privileges on a table must have any of the following privileges:

- DBA privilege
- Ownership of the table
- All the specified privileges on the table, granted with the WITH GRANT OPTION clause

SQL Compliance	SQL-92, ODBC Core SQL grammar. Extensions: ALTER, INDEX, RESOURCE, DBA privileges
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	GRANT

2.15 SELECT

Description

Selects the specified column values from one or more rows contained in the table(s) specified in the FROM clause. The selection of rows is restricted by the WHERE clause. The temporary table derived through the clauses of a select statement is called a result table.

The format of the SELECT statement is a query expression with optional ORDER BY and FOR UPDATE clauses. For more detail on query expressions, see “1.5 Query Expressions” on page 1-14.

Syntax

```
select_statement ::
   query_expression
   ORDER BY { expr | posn } [ COLLATE collation_name ] [ ASC | DESC ]
[ , { expr | posn } [ COLLATE collation_name ] [ASC | DESC] ,... ]
   FOR UPDATE [ OF [table].column_name, ... ] [ NOWAIT ];
query_expression ::
    query_specification
|   query_expression set_operator query_expression
|   ( query_expression )

set_operator ::
     {  UNION [ ALL ]  |  INTERSECT  |  MINUS }
query_specification ::
SELECT [ALL | DISTINCT]
        {
           *
        |  { table_name | alias } . * [, { table_name | alias } . * ] …
        |  expr [ [ AS ] [ ' ] column_title [ ' ] ] [, expr [ [ AS ] [
' ] column_title [ ' ] ] ] ...
        }
FROM table_ref [ { dharma ORDERED } ] [ , table_ref [ { dharma ORDERED
} ] …
[ WHERE search_condition ]
[ GROUP BY [table.]column_name [ COLLATE collation-name ]
                  [, [table.]column_name [ COLLATE collation-name ] ]
...
[ HAVING search_condition ]

table_ref ::
           table_name [ AS ] [ alias [ ( column_alias [ , … ] ) ] ]
       |  ( query_expression ) [ AS ] alias [ ( column_alias [ , … ] )
]
        |  [ ( ] joined_table [ ) ]

joined_table ::
```

```
            table_ref CROSS JOIN table_ref
          |  table_ref [ INNER | LEFT [ OUTER ] ] JOIN table_ref ON
search_condition
```

Arguments

query_expression
See "1.5 Query Expressions" on page 1-14.

ORDER BY clause
See *Order By Clause*, next section.

FOR UPDATE clause
See *Update* section on page 2-33.

Authorization

The user executing this statement must have any of the following privileges:

- DBA privilege
- SELECT permission on all the tables/views referred to in the *query_expression*.

SQL Compliance	SQL-92. Extensions: FOR UPDATE clause. ODBC Extended SQL grammar.
Environment	Embedded SQL (within DECLARE), interactive SQL, ODBC applications
Related Statements	Query Expressions, DECLARE CURSOR, OPEN, FETCH, CLOSE

2.15.1 ORDER BY Clause

Description

The ORDER BY clause specifies the sorting of rows retrieved by the SELECT statement. SQL does not guarantee the sort order of rows unless the SELECT statement includes an ORDER BY clause.

Syntax

```
ORDER BY { expr | posn } [ COLLATE collation_name ] [ ASC | DESC ]
     [ , { expr | posn } [ COLLATE collation_name ] [ASC | DESC] ,... ]
```

Notes

- Ascending order is the default ordering. The descending order will be used only if the keyword DESC is specified for that column.
- Each *expr* is an expression of one or more columns of the tables specified in the FROM clause of the SELECT statement. Each *posn* is a number identifying the column position of the columns being selected by the SELECT statement.
- The selected rows are ordered on the basis of the first *expr* or *posn* and if the values are the same then the second *expr* or *posn* is used in the ordering.

- The ORDER BY clause if specified should follow all other clauses of the SELECT statement.
- A query expression followed by an optional ORDER BY clause can be specified. In such a case, if the query expression contains set operators, then the ORDER BY clause can specify only the positions. For example:

```
-- Get a merged list of customers and suppliers
-- sorted by their name.
     (SELECT name, street, state, zip
     FROM customer
     UNION
     SELECT name, street, state, zip
     FROM supplier)
     ORDER BY 1 ;
```

- If *expr* or *posn* refers to a character column, the reference can include an optional COLLATE clause. The COLLATE clause specifies a collation sequence supported by the underlying storage system. Sea "1.4.1.1 Specifying the Character Set for Character Data Types" on page 1-6 for notes on character sets and collations. See the documentation for your underlying storage system for details on any supported collations.)

Example

```
SELECT name, street, city, state, zip
     FROM customer
     ORDER BY name ;
```

2.15.2 FOR UPDATE Clause

Description

The FOR UPDATE clause specifies update intention on the rows selected by the SELECT statement.

Syntax

```
FOR UPDATE [ OF [table].column_name, ... ] [ NOWAIT ]Notes
```

- If FOR UPDATE clause is specified, WRITE locks are acquired on all the rows selected by the SELECT statement.
- If NOWAIT is specified, an error is returned when a lock cannot be acquired on a row in the selection set because of the lock held by some other transaction. Otherwise, the transaction would wait until it gets the required lock or until it times out waiting for the lock.

2.16 SET SCHEMA

Description

SET SCHEMA specifies a new default qualifier for database object names. (Database objects include tables, indexes, views, synonyms, procedures, and triggers.)

When you connect to a database with a particular user name, that name becomes the default qualifier for database object names. This means you do not have to qualify references to tables, for instance, that were created under the same user name. However, you must qualify references to all other tables with the user name of the user who created them.

SET SCHEMA allows you to change the user name that SQL uses as the default qualifier for database object names. The name specified in SET SCHEMA becomes the new default qualifier for object names.

Note: SET SCHEMA does not change your user name or affect authentication. It only changes the default qualifier.

Syntax

```
SET SCHEMA ' qualifier_name ' ;
```

Arguments

`' qualifier_name '`
The new qualifier name, enclosed in single quotation marks.

Notes

- SET SCHEMA does not check whether *qualifier_name* is a valid user name.
- Metadata for objects created without an explicit qualifier will show *qualifier_name* as the owner.
- SET SCHEMA does not start or end a transaction.

Examples

The following interactive SQL example shows changing the default qualifier through SET SCHEMA. The example:

- Invokes ISQL as the user systpe, the owner of the system catalog tables
- Queries the *systables* catalog tables as *systpe*
- Uses SET SCHEMA to change the default qualifier to *fred*
- Creates a table and queries *systables* to show that the newly-created table is owned by *fred*

```
ISQL> -- What is the user name for the current connection?
ISQL> select user() from syscalctable;
SYSTPE
------
```

```
systpe
1 record selected
ISQL> -- Show the name and owner of non-system tables:
ISQL> select tbl, owner from systables where tbltype <> 'S';
TBL                             OWNER
---                              -----
t1                               systpe
test                            systpe
test                            dharma
3 records selected
ISQL> set schema 'fred';
ISQL> create table freds_table (c1 int);
ISQL> create index freds_table_ix on freds_table (c1);
ISQL> select tbl, owner from systables where tbltype <> 'S';
select tbl, owner from systables where tbltype <> 'S';
                                  *
error(-20005): Table/View/Synonym not found
ISQL> -- Oops! Must now qualify references to the systpe-owned
tables:
ISQL> select tbl, owner from systpe.systables where tbltype <>
'S';
TBL                             OWNER
---                              -----
t1                               systpe
test                            systpe
test                            dharma
freds_table                      fred
4 records selected
```

Authorization

None.

SQL Compliance	SQL-92
Environment	Embedded SQL and interactive
Related Statements	None

2.17 UPDATE

Description

Updates the columns of the specified table with the given values that satisfy the *search_condition*.

Syntax

```
UPDATE table_name
     SET assignment, assignment, ...
     [ WHERE search_condition ]
assignment ::
          column = { expr | NULL }
          | ( column, column, ... ) = ( expr, expr, ... )
          | ( column, column, ... ) = ( query_expression )
```

Arguments

If the optional WHERE clause is specified, then only rows that satisfy the *search_condition* are updated. If the WHERE clause is not specified then all rows of the table are updated.

The expressions in the SET clause are evaluated for each row of the table if they are dependent on the columns of the target table.

If a query expression is specified on the right hand side for an assignment, the number of expressions in the first SELECT clause of the query expression must be the same as the number of columns listed on the left hand side of the assignment.

If a query expression is specified on the right hand side for an assignment, the query expression must return one row.

If a table has check constraints and if the columns to be updated are part of a check expression, then the check expression is evaluated. If the result of evaluation is FALSE, the UPDATE statement fails.

If a table has primary/candidate keys and if the columns to be updated are part of the primary/candidate key, a check is made as to whether there exists any corresponding row in the referencing table. If so, the UPDATE operation fails.

Examples

```
UPDATE orders
          SET qty = 12000
          WHERE order_no = 1001 ;
UPDATE orders
          SET (product) =
                    (SELECT item_name
                     FROM items
                     WHERE item_no = 2401
```

```
                    )
          WHERE order_no = 1002 ;
UPDATE orders
          SET (amount) = (2000 * 30)
          WHERE order_no = 1004 ;
UPDATE orders
          SET (product, amount) =
                              (SELECT item_name, price * 30
                               FROM items
                               WHERE item_no = 2401
                              )
          WHERE order_no = 1002 ;
```

Authorization

The user executing this statement must have:

- DBA privilege.
- UPDATE privilege on all the specified columns of the target table and SELECT privilege on all the other tables referred to in the statement.

SQL Compliance	SQL-92, ODBC Extended SQL grammar. Extensions: assignments of the form (column, column, ...) = (expr, expr, ...)
Environment	Embedded SQL, interactive SQL, ODBC applications
Related Statements	SELECT, OPEN, FETCH, search conditions, query expressions

Appendix A

Reserved Words

A.1 RESERVED WORDS

Reserved words are keywords you can use as identifiers in SQL statements if you delimit them with double quotation marks. If you use keywords without delimiting them, the statement generates one of the following errors:

```
error(-20003): Syntax error
error(-20049): Keyword used for a name
```

The following table lists reserved words. The list is alphabetic and reads left to right.

Table A-1: Reserved Words

A	ABS	ACOS	ADD
ADD_MONTHS	AFTER	ALL	ALTER
AN	AND	ANY	ARRAY
AS	ASC	ASCII	ASIN
ATAN	ATAN2	AVG	BEFORE
BEGIN	BETWEEN	BIGINT	BINARY
BIND	BINDING	BIT	BY
CALL	CASCADE	CASE	CAST
CEILING	CHAR	CHAR_LENGTH	CHARACTER
CHARACTER_LENGTH	CHARTOROWID	CHECK	CHR
CLEANUP	CLOSE	CLUSTERED	COALESCE
COLGROUP	COLLATE	COLUMN	COMMIT
COMPLEX	COMPRESS	CONCAT	CONNECT
CONSTRAINT	CONTAINS	CONTINUE	CONVERT
COS	COUNT	CREATE	CROSS
CURDATE	CURRENT	CURSOR	CURTIME
CVAR	DATABASE	DATAPAGES	DATE
DAYNAME	DAYOFMONTH	DAYOFWEEK	DAYOFYEAR
DB_NAME	DBA	DEC	DECIMAL
DECLARATION	DECLARE	DECODE	DEFAULT

Table A-1: Reserved Words

DEFINITION	DEGREES	DELETE	DESC
DESCRIBE	DESCRIPTOR	DHTYPE	DIFFERENCE
DISTINCT	DOUBLE	DROP	EACH
ELSE	END	END	ESCAPE
EXCLUSIVE	EXEC	EXECUTE	EXISTS
EXIT	EXP	EXPLICIT	FETCH
FIELD	FILE	FLOAT	FLOOR
FOR	FOREIGN	FOUND	FROM
FULL	GO	GOTO	GRANT
GREATEST	GROUP	HASH	HAVING
HOUR	IDENTIFIED	IFNULL	IMMEDIATE
IN	INDEX	INDEXPAGES	INDICATOR
INITCAP	INNER	INOUT	INPUT
INSERT	INSTR	INT	INTEGER
INTERFACE	INTERSECT	INTO	IS
JOIN	KEY	LAST_DAY	LCASE
LEAST	LEFT	LEFT	LENGTH
LIKE	LINK	LIST	LOCATE
LOCK	LOG	LOG10	LONG
LONG	LOWER	LPAD	LTRIM
LVARBINARY	LVARCHAR	MAIN	MAX
METADATA_ONLY	MIN	MINUS	MINUTE
MOD	MODE	MODIFY	MONEY
MONTH	MONTHNAME	MONTHS_BETWEEN	NATIONAL
NATURAL	NCHAR	NEWROW	NEXT_DAY
NOCOMPRESS	NOT	NOW	NOWAIT
NULL	NULLIF	NULLVALUE	NUMBER
NUMERIC	NVL	OBJECT_ID	ODBC_CONVERT
ODBCINFO	OF	OLDROW	ON
OPEN	OPTION	OR	ORDER
OUT	OUTER	OUTPUT	PCTFREE
PI	POWER	PRECISION	PREFIX

Table A-1: Reserved Words

PREPARE	PRIMARY	PRIVILEGES	PROCEDURE
PUBLIC	QUARTER	RADIANS	RAND
RANGE	RAW	REAL	RECORD
REFERENCES	REFERENCING	RENAME	REPEAT
REPLACE	RESOURCE	RESTRICT	RESULT
RETURN	REVOKE	RIGHT	RIGHT
ROLLBACK	ROW	ROWID	ROWIDTOCHAR
ROWNUM	RPAD	RTRIM	SEARCHED_CASE
SECOND	SECTION	SELECT	SERVICE
SET	SHARE	SHORT	SIGN
SIMPLE_CASE	SIN	SIZE	SMALLINT
SOME	SOUNDEX	SPACE	SQL
SQL_BIGINT	SQL_BINARY	SQL_BIT	SQL_CHAR
SQL_DATE	SQL_DECIMAL	SQL_DOUBLE	SQL_FLOAT
SQL_INTEGER	SQL_LONGVARBINARY	SQL_LONGVARCHAR	SQL_NUMERIC
SQL_REAL	SQL_SMALLINT	SQL_TIME	SQL_TIMESTAMP
SQL_TINYINT	SQL_TSI_DAY	SQL_TSI_FRAC_SECOND	SQL_TSI_HOUR
SQL_TSI_MINUTE	SQL_TSI_MONTH	SQL_TSI_QUARTER	SQL_TSI_SECOND
SQL_TSI_WEEK	SQL_TSI_YEAR	SQL_VARBINARY	SQL_VARCHAR
SQLERROR	SQLWARNING	SQRT	START
STATEMENT	STATISTICS	STOP	STORAGE_ATTRIBUTES
STORAGE_MANAGER	STORE_IN_DHARMA	SUBSTR	SUBSTRING
SUFFIX	SUM	SUSER_NAME	SYNONYM
SYSDATE	SYSTIME	SYSTIMES-TAMP	TABLE
TAN	THEN	TIME	TIMEOUT
TIMESTAMP	TIMESTAMPADD	TIMESTAMP-DIFF	TINYINT
TO	TO_CHAR	TO_DATE	TO_NUMBER
TO_TIME	TO_TIMESTAMP	TPE	TRANSACTION

Table A-1: Reserved Words

TRANSLATE	TRIGGER	TYPE	UCASE
UID	UNION	UNIQUE	UNSIGNED
UPDATE	UPPER	USER	USER_ID
USER_NAME	USING	UUID	VALUES
VARBINARY	VARCHAR	VARIABLES	VARYING
VERSION	VIEW	WEEK	WHEN
WHENEVER	WHERE	WITH	WORK
YEAR			

Appendix B

Error Messages

B.1 OVERVIEW

This appendix lists the error messages generated by the various components of the Dharma SDK.

You can receive error messages not only from the SQL engine, but also from underlying storage systems, including the storage system Dharma supplies for use as a data dictionary.

Storage systems on which the Dharma SDK is implemented will likely generate their own error messages. See the documentation for your storage system for details.

B.2 ERROR CODES, SQLSTATE VALUES, MESSAGES

The following table lists the Dharma SDK error messages, ordered by error code number, and shows the corresponding SQLSTATE values for each message.

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
00000	00000	Successful completion	***status okay
100L	02000	no data	**sql not found.
10002	02503	No data	Tuple not found for the specified TID.
10012	n0n12	flag	ETPL_SCAN_EOP flag is set.
10013	02514	No data	No more records to be fetched.
10100	2150b	Cardinality violation	Too many fields specified.
10101	02701	No data	No more records exists.
10102	5050c	Dharma/SQL rds error	Duplicate record specified.
10104	22505	Data exception	Field size is too high.
10106	m0m06	Dharma/SQL rss error	Specified index method is not supported.
10107	n0n07	flag	EIX_SCAN_EOP flag is set.
10108	2350i	Integrity constraint	Duplicate primary /index key value.
10301	m030a	Dharma/SQL rss error	Table is locked and LCKF_NOWAIT.
10400	22501	Data exception	Invalid file size for alter log statement.
10920	22521	Data exception	Already existing value specified.

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
11100	5050b	Dharma/SQL rds error	Invalid transaction id.
11102	5050d	Dharma/SQL rds error	TDS area specified is not found.
11103	50504	Dharma/SQL rds error	TDS not found for binding.
11104	50505	Dharma/SQL rds error	Transaction aborted.
11105	50506	Dharma/SQL rds error	Active Transaction error.
11109	50510	Dharma/SQL rds error	Invalid Transaction handle.
11111	50912	Dharma/SQL rds error	Invalid isolation level.
11300	m0m00	Dharma/SQL rss error	Specified INFO type is not supported.
11301	m0m01	Dharma/SQL rss error	Specified index type is not supported.
15001	60601	dharma/SQL ff errors	FF- File IO error
15002	60602	dharma/SQL ff errors	FF- No more records
15003	42603	Access violation error	FF- Table already exists
15004	22604	Data exception	FF- Invalid record number
15005	60605	dharma/SQL ff errors	FF- Record deleted
15006	60606	dharma/SQL ff errors	FF- Invalid type
15007	60607	dharma/SQL ff errors	FF- Duplicate value
15008	08608	Connection exception	FF- Database exists
15009	08609	Connection exception	FF- No database found
15010	60610	dharma/SQL ff errors	FF- Version mis-match
15011	60611	dharma/SQL ff errors	FF- Virtual file cache exceeded
15012	60612	dharma/SQL ff errors	FF- Physical file open error
15013	60613	dharma/SQL ff errors	FF- Corrupt virtual file handle
15014	22614	Data exception	FF- Overflow error
15021	60615	dharma/SQL ff errors	FF- dbm_calls not implemented
16001	22701	Data exception	MM- No data block
16002	70702	dharma/SQL MM errors	MM- Bad swap block
16003	70703	dharma/SQL MM errors	MM- No cache block
16004	22704	Data exception	MM- Invalid row number
16005	70705	dharma/SQL MM errors	MM- Invalid cache block
16006	70706	dharma/SQL MM errors	MM- Bad swap file
16007	70707	dharma/SQL MM errors	MM- Row too big

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
16008	70708	dharma/SQL MM errors	MM- Array initialized
16009	70709	dharma/SQL MM errors	MM- Invalid chunk number
16010	70710	dharma/SQL MM errors	MM- Can't create table
16011	70711	dharma/SQL MM errors	MM- Can't alter table
16012	70712	dharma/SQL MM errors	MM- Can't drop table
16020	70713	dharma/SQL MM errors	MM- TPL ctor error
16021	70714	dharma/SQL MM errors	MM- Insertion error
16022	70715	dharma/SQL MM errors	MM- Deletion error
16023	70716	dharma/SQL MM errors	MM- Updation error
16024	70717	dharma/SQL MM errors	MM- Fetching error
16025	70718	dharma/SQL MM errors	MM- Sorting error
16026	70719	dharma/SQL MM errors	MM- Printing error
16027	70720	dharma/SQL MM errors	MM- TPLSCAN ctor error
16028	70721	dharma/SQL MM errors	MM- Scan fetching error
16030	70722	dharma/SQL MM errors	MM- Can't create index
16031	70723	dharma/SQL MM errors	MM- Can't drop index
16032	70724	dharma/SQL MM errors	MM- IXSCAN ctor error
16033	70725	dharma/SQL MM errors	MM- IX ctor error
16034	70726	dharma/SQL MM errors	MM- IX deletion error
16035	70727	dharma/SQL MM errors	MM- IX appending error
16036	70728	dharma/SQL MM errors	MM- IX insertion error
16037	70729	dharma/SQL MM errors	MM- IX scan fetching error
16040	70730	dharma/SQL MM errors	MM- Begin transaction
16041	70731	dharma/SQL MM errors	MM- Commit transaction
16042	40000	Transaction rollback	***MM- Rollback transaction
16043	70732	dharma/SQL MM errors	MM- Mark point
16044	70733	dharma/SQL MM errors	MM- Rollback savepoint
16045	70734	dharma/SQL MM errors	MM- Set & Get isolation
16050	70735	dharma/SQL MM errors	MM- TID to char
16051	70736	dharma/SQL MM errors	MM- char to TID
20000	50501	dharma/SQL rds error	SQL internal error

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20001	50502	dharma/SQL rds error	Memory allocation failure
20002	50503	dharma/SQL rds error	Open database failed
20003	2a504	Syntax error	Syntax error
20004	28505	Invalid auth specs	User not found
20005	22506	Data exception	Table/View/Synonym not found
20006	22507	Data exception	Column not found/specified
20007	22508	Data exception	No columns in table
20008	22509	Data exception	Inconsistent types
20009	22510	Data exception	Column ambiguously specified
20010	22511	Data exception	Duplicate column specification
20011	22512	Data exception	Invalid length
20012	22513	Data exception	Invalid precision
20013	22514	Data exception	Invalid scale
20014	22515	Data exception	Missing input parameters
20015	22516	Data exception	Subquery returns multiple rows
20016	22517	Data exception	Null value supplied for a mandatory (not null) column
20017	22518	Data exception	Too many values specified
20018	22519	Data exception	Too few values specified
20019	50520	dharma/SQL rds error	Can not modify table referred to in subquery
20020	42521	Access rule violation	Bad column specification for group by clause
20021	42522	Access rule violation	Non-group-by expression in having clause
20022	42523	Access rule violation	Non-group-by expression in select clause
20023	42524	Access rule violation	Aggregate function not allowed here
20024	0a000	feature not supported	Sorry, operation not yet implemented
20025	42526	Access rule violation	Aggregate functions nested
20026	50527	dharma/SQL rds error	Too many table references
20027	42528	Access rule violation	Bad field specification in order by clause

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20028	50529	dharma/SQL rds error	An index with the same name already exists
20029	50530	dharma/SQL rds error	Index referenced not found
20030	22531	Data exception	Table space with same name already exists
20031	50532	dharma/SQL rds error	Cluster with same name already exists
20032	50533	dharma/SQL rds error	No cluster with this name
20033	22534	Data exception	Tablespace not found
20034	50535	dharma/SQL rds error	Bad free percentage specification
20035	50536	dharma/SQL rds error	At least column spec or null clause should be specified
20036	07537	dynamic sql error	Statement not prepared
20037	24538	Invalid cursor state	Executing select statement
20038	24539	Invalid cursor state	Cursor not closed
20039	24540	Invalid cursor state	Open for non select statement
20040	24541	Invalid cursor state	Cursor not opened
20041	22542	Data exception	Table/View/Synonym already exists
20042	2a543	Syntax error	Distinct specified more than once in query
20043	50544	dharma/SQL rds error	Tuple size too high
20044	50545	dharma/SQL rds error	Array size too high
20045	08546	Connection exception	File does not exist or not accessible
20046	50547	dharma/SQL rds error	Field value not null for some tuples
20047	42548	Access rule violation	Granting to self not allowed
20048	42549	Access rule violation	Revoking for self not allowed
20049	22550	Data exception	Keyword used for a name
20050	21551	Cardinality violation	Too many fields specified
20051	21552	Cardinality violation	Too many indexes on this table
20052	22553	Data exception	Overflow/Underflow error
20053	08554	Connection exception	Database not opened
20054	08555	Connection exception	Database not specified or improperly specified

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20055	08556	Connection exception	Database not specified or Database not started
20056	28557	Invalid auth specs	No DBA access rights
20057	28558	Invalid auth specs	No RESOURCE privileges
20058	40559	Transaction rollback	Executing SQL statement for an aborted transaction
20059	22560	Data exception	No files in the table space
20060	22561	Data exception	Table not empty
20061	22562	Data exception	Input parameter size too high
20062	42563	Syntax error	Full pathname not specified
20063	50564	dharma/SQL rds error	Duplicate file specification
20064	08565	Connection exception	Invalid attach type
20065	26000	Invalid SQL statement name	Invalid statement type
20066	33567	Invalid SQL descriptor name	Invalid sqlda
20067	08568	Connection exception	More than one database can't be attached locally
20068	42569	Syntax error	Bad arguments
20069	33570	Invalid SQL descriptor name	SQLDA size not enough
20070	33571	Invalid SQL descriptor name	SQLDA buffer length too high
20071	42572	Access rule violation	Specified operation not allowed on the view
20072	50573	dharma/SQL rds error	Server is not allocated
20073	2a574	Access rule violation	View query specification for view too long
20074	2a575	Access rule violation	View column list must be specified as expressions are given
20075	21576	Cardinality violation	Number of columns in column list is less than in select list
20076	21577	Cardinality violation	Number of columns in column list is more than in select list
20077	42578	Access rule violation	Check option specified for non-insert-able view
20078	42579	Access rule violation	Given SQL statement is not allowed on the view
20079	50580	dharma/SQL rds error	More tables cannot be created.

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20080	44581	Check option violation	View check option violation
20081	22582	Data exception	No of expressions projected on either side of set-op don't match
20082	42583	Access rule violation	Column names not allowed in order by clause for this statement
20083	42584	Access rule violation	Outerjoin specified on a complex predicate
20084	42585	Access rule violation	Outerjoin specified on a sub_query
20085	42586	Access rule violation	Invalid Outerjoin specification
20086	42587	Access rule violation	Duplicate table constraint specification
20087	21588	Cardinality violation	Column count mismatch
20088	28589	Invalid auth specs	Invalid user name
20089	22590	Data exception	System date retrieval failed
20090	42591	Access rule violation	Table columnlist must be specified as expressions are given
20091	2a592	Access rule violation	Query statement too long.
20092	2d593	Invalid transaction termination	No tuples selected by the subquery for update
20093	22594	Data exception	Synonym already exists
20094	hz595	Remote database access	Database link with same name already exists
20095	hz596	Remote database access	Database link not found
20096	08597	Connection exception	Connect String not specified/incorrect
20097	hz598	Remote database access	Specified operation not allowed on a remote table
20098	22599	Data exception	More than one row selected by the query
20099	24000	Invalid cursor state	Cursor not positioned on a valid row
20100	4250a	Access rule violation	Subquery not allowed here
20101	2350b	Integrity constraint	No references for the table
20102	2350c	Integrity constraint	Primary/Candidate key column defined null
20103	2350d	Integrity constraint	No matching key defined for the referenced table

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20104	2350e	Integrity constraint	Keys in reference constraint incompatible
20105	5050f	dharma/SQL rds error	Statement not allowed in readonly isolation level
20106	2150g	Cardinality violation	Invalid ROWID
20107	hz50h	Remote database access	Remote Database not started
20108	0850i	Connection exception	Remote Network Server not started.
20109	hz50j	Remote database access	Remote Database Name not valid
20110	0850k	Connection exception	TCP/IP Remote HostName is unknown.
20114	33002	Invalid SQL descriptor name	Fetched Value NULL & indicator var not defined
20115	5050l	dharma/SQL rds error	References to the table/record present
20116	2350m	Integrity constraint	Constraint violation
20117	2350n	Integrity constraint	Table definition not complete
20118	4250o	Access rule violation	Duplicate constraint name
20119	2350p	Integrity constraint	Constraint name not found
20120	22000	Data exception	**use of reserved word
20121	5050q	dharma/SQL rds error	permission denied
20122	5050r	dharma/SQL rds error	procedure not found
20123	5050s	dharma/SQL rds error	invalid arguments to procedure
20124	5050t	dharma/SQL rds error	Query conditionally terminated
20125	0750u	Dynamic sql-error	Number of open cursors exceeds limit
20126	34000	Invalid cursor name	***Invalid cursor name
20127	07001	Dynamic sql-error	Bad parameter specification for the statement
20128	2250x	Data Exception	Numeric value out of range.
20129	2250y	Data Exception	Data truncated.
20130	40000	Dharma/SQL rds error	Rolled back transaction.
20131	50522	dharma/SQL rds error	Value for Parameter marker is Missing
20132	5050u	dharma/SQL rds error	Revoke failed because of restrict
20133	0a001	Feature not supported	Feature not supported

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20133	0a000	feature not supported	Sorry, Feature not supported in this Edition.
20134	5050v	dharma/SQL rds error	Invalid long datatype column references
20135	5050x	dharma/SQL rds error	Contains operator is not supported in this context
20135	m0m01	dharma/SQL diagnostics error	diagnostics statement failed.
20136	5050z	dharma/SQL rds error	Contains operator is not supported for this datatype
20137	50514	dharma/SQL rds error	Index is not defined or does not support CONTAINS
20138	50513	dharma/SQL rds error	Index on long fields requires that it can push down only CONTAINS
20140	50512	dharma/SQL rds error	Procedure already exists
20141	85001	dharma/SQL Stored procedure Compilation	Error in Stored Procedure Compilation.
20142	86001	dharma/SQL Stored procedure Execution	Error in Stored Procedure Execution.
20143	86002	dharma/SQL Stored procedure Execution	Too many recursions in call procedure.
20144	86003	dharma/SQL Stored procedure Execution	Null value fetched.
20145	86004	dharma/SQL Stored procedure Execution	Invalid field reference.
20146	86005	dharma/SQL Triggers	Trigger with this name already exists.
20147	86006	dharma/SQL Triggers	Trigger with this name does not exist.
20148	86007	dharma/SQL Triggers	Trigger Execution Failed.
20150	50512z	dharma/SQL rds error	view manager ID is not found.
20151	50515	dharma/SQL rds error	Cannot drop all columns; use DROP TABLE instead
20152	50516	dharma/SQL rds error	Cannot preallocate memory for n cache items
20153	50517	dharma/SQL rds error	Tree not present in cache; search failed
20154	50518	dharma/SQL rds error	Statement cache insert failed
20155	50519	dharma/SQL rds error	Environment variable used for the creation of multi component index is not set properly.

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20156	50521	dharma/SQL rds error	No SQL statement
20157	50523	dharma/SQL rds error	Invalid JSP/T method sequence.
20159	28590	Invalid auth specs	Can't REVOKE privileges, you did not grant
20160	28591	Feature not supported	JSPT Feature not supported
20164	2a565	Syntax error	Escape character must be character string of length 1.
20165	2a566	Syntax error	Missing or illegal character following the escape character
20211	22800	Data exception	Remote procedure call error
20212	08801	Connection exception	SQL client bind to daemon failed
20213	08802	Connection exception	SQL client bind to SQL server failed
20214	08803	Connection exception	SQL NETWORK service entry is not available
20215	08804	Connection exception	Invalid TCP/IP hostname
20216	hz805	Remote database access	Invalid remote database name
20217	08806	Connection exception	network error on server
20218	08807	Connection exception	Invalid protocol
20219	2e000	Invalid connection name	***Invalid connection name
20220	08809	Connection exception	Duplicate connection name
20221	08810	Connection exception	No active connection
20222	08811	Connection exception	No environment defined database
20223	08812	Connection exception	Multiple local connections
20224	08813	Connection exception	Invalid protocol in connect_string
20225	08814	Connection exception	Exceeding permissible number of connections
20226	80815	dharma/SQL snw errors	Bad database handle
20227	08816	Connection exception	Invalid host name in connect string
20228	28817	Invalid auth specs	Access denied (Authorization failed)
20229	22818	Data exception	Invalid date value
20230	22819	Data exception	Invalid date string
20231	22820	Data exception	Invalid number strings
20232	22821	Data exception	Invalid number string

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
20233	22822	Data exception	Invalid time value
20234	22523	Data exception	Invalid time string
20235	22007	Data exception	Invalid time stamp string
20236	22012	Data exception	Division by zero attempted
20238	22615	Data exception	Error in format type
20239	2c000	Invalid character set name	Invalid character set name specified.
20240	5050y	dharma/SQL rds errors	Invalid collation name specified.
20241	08815	Connection Exception	Service in use.
20242	22008	Data exception	Invalid timestamp
20248	22601	Data exception	Identifier too long
20250	22603	Data exception	Invalid Identifier
20251	22604	Syntax error	Quoted string not properly terminated
20252	22605	Syntax error	Missing double quote in Identifier
20300	90901	dharma/DBS errors	Column group column doesn't exist
20301	90902	dharma/DBS errors	Column group column already specified
20302	90903	dharma/DBS errors	Column group name already specified
20303	90904	dharma/DBS errors	Column groups haven't covered all columns
20304	90905	dharma/DBS errors	Column groups are not implemented in Dharma storage
23000	22563	dharma/SQL Data exception	Table create returned invalid table id
23001	22564	dharma/SQL Data exception	Index create returned invalid index id
25001	i0i01	dharma/SQL odbc integrator	Operation is valid only on a linked table
25002	i0i02	dharma/SQL odbc integrator	Operation not allowed on a linked table
25003	i0i03	dharma/SQL odbc integrator	Copy object exists
25004	i0i04	dharma/SQL odbc integrator	Unknown copy object
25005	i0i05	dharma/SQL odbc integrator	Dropping table failed
25006	i0i06	dharma/SQL odbc integrator	Bad copy sql statement
25007	i0i07	dharma/SQL odbc integrator	Unknown data type
25008	i0i08	dharma/SQL odbc integrator	Bad insert statement

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
25009	i0i09	dharma/SQL odbc integrator	Fetch operation failed
25010	i0i10	dharma/SQL odbc integrator	Insert operation failed
25011	i0i11	dharma/SQL odbc integrator	Operation not started
25012	i0i12	dharma/SQL odbc integrator	Operation marked for abort
25013	i0i13	dharma/SQL odbc integrator	Commit operation failed
25014	i0i14	dharma/SQL odbc integrator	Create table failed
25015	i0i15	dharma/SQL odbc integrator	Bad sync sql statement
25016	i0i16	dharma/SQL odbc integrator	Sync object exists
25017	i0i17	dharma/SQL odbc integrator	Create sync object failed
25018	i0i18	dharma/SQL odbc integrator	Create copy object failed
25019	i0i19	dharma/SQL odbc integrator	Unknown sync object
25020	i0i20	dharma/SQL odbc integrator	Illegal column name
25021	i0i21	dharma/SQL odbc integrator	Duplicate column name
25022	i0i22	dharma/SQL odbc integrator	Install failure
25023	i0i23	dharma/SQL odbc integrator	Invalid sync mode
25024	i0i24	dharma/SQL odbc integrator	Download or snapshot table missing
25025	i0i25	dharma/SQL odbc integrator	Upload table missing
25026	i0i26	dharma/SQL odbc integrator	Update operation failed
25027	i0i27	dharma/SQL odbc integrator	Delete operation failed
25028	i0i28	dharma/SQL odbc integrator	Close cursor failed
25029	i0i29	dharma/SQL odbc integrator	No primary key
25030	i0i30	dharma/SQL odbc integrator	Missing row
25031	i0i31	dharma/SQL odbc integrator	Bad primary key
25032	i0i32	dharma/SQL odbc integrator	Update contention
25033	i0i33	dharma/SQL odbc integrator	Link table failed
25034	i0i34	dharma/SQL odbc integrator	Unlink table failed
25035	i0i35	dharma/SQL odbc integrator	Link data source failed
25036	i0i36	dharma/SQL odbc integrator	Unlink data source failed
25037	i0i37	dharma/SQL odbc integrator	Integrator internal error
25038	i0i38	dharma/SQL odbc integrator	Operation already started
25039	i0i39	dharma/SQL odbc integrator	Opening of copy sql stmt failed

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
25040	i0i40	dharma/SQL odbc integrator	sync object failed
25041	i0i41	dharma/SQL odbc integrator	Dropping copy object failed
25042	i0i42	dharma/SQL odbc integrator	Closing copy sql stmt failed
25043	i0i43	dharma/SQL odbc integrator	Failure to update metadata timestamp
25101	j0j01	dharma/SQL odbc trans layer	SQLAllocEnv failed
25102	j0j02	dharma/SQL odbc trans layer	SQLAllocConnect failed
25103	j0j03	dharma/SQL odbc trans layer	SQLConnect failed
25104	j0j04	dharma/SQL odbc trans layer	SQLGetConnectOption failed
25105	j0j05	dharma/SQL odbc trans layer	SQLSetConnectOption failed
25106	j0j06	dharma/SQL odbc trans layer	Failed to map stmt handle to UUID
25107	j0j07	dharma/SQL odbc trans layer	SQLSetParam failed
25108	j0j08	dharma/SQL odbc trans layer	SQLDisconnect failed
25109	j0j09	dharma/SQL odbc trans layer	SQLExecute failed
25110	j0j10	dharma/SQL odbc trans layer	SQLRowCount failed
25111	j0j11	dharma/SQL odbc trans layer	SQLSetParam failed
25112	j0j12	dharma/SQL odbc trans layer	SQLBindCol failed
25113	j0j13	dharma/SQL odbc trans layer	SQLPrepare failed
25114	j0j14	dharma/SQL odbc trans layer	SQLResultCols failed
25115	j0j15	dharma/SQL odbc trans layer	SQLDescribeCol failed
25116	j0j16	dharma/SQL odbc trans layer	SQLFreeStmt failed
25117	j0j17	dharma/SQL odbc trans layer	SQLFetch failed
25118	j0j18	dharma/SQL odbc trans layer	SQLTransact failed
25119	j0j19	dharma/SQL odbc trans layer	SQLAllocStmt failed
25120	j0j20	dharma/SQL odbc trans layer	SQLTables failed
25121	j0j21	dharma/SQL odbc trans layer	SQLColumns failed
25122	j0j22	dharma/SQL odbc trans layer	SQLStatistics failed
25123	j0j23	dharma/SQL odbc trans layer	ODBC Driver interface mismatch
25124	j0j24	dharma/SQL odbc trans layer	ODBC Driver metadata exceeds storage limits
25125	j0j25	dharma/SQL odbc trans layer	SQLGetInfo failed
25126	j0j26	dharma/SQL odbc trans layer	operation not allowed on the read-only database

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
25127	j0j27	dharma/SQL odbc trans layer	cannot update views-with-check-option on remote tables
25128	j0j28	dharma/SQL odbc trans layer	query terminated as max row limit exceeded for a remote table
25131	j0j29	dharma/SQL odbc trans layer	unable to read column info from remote table
26001	08001	dharma/SQL JDBC errors	Unable to connect to data source
26002	08003	dharma/SQL JDBC errors	Connection does not exist
26003	08S01	dharma/SQL JDBC errors	Communication link failure
26004	08597	dharma/SQL JDBC errors	Connect String not specified/incorrect
26005	0850i	dharma/SQL JDBC errors	Remote Network Server not started
26006	0850k	dharma/SQL JDBC errors	TCP/IP Remote HostName is unknown
26007	08801	dharma/SQL JDBC errors	SQL client bind to daemon failed
26008	08802	dharma/SQL JDBC errors	SQL client bind to SQL server failed
26009	08803	dharma/SQL JDBC errors	SQL Network service entry is not available
26010	08804	dharma/SQL JDBC errors	Invalid TCP/IP hostname
26011	08806	dharma/SQL JDBC errors	Network error on server
26012	08807	dharma/SQL JDBC errors	Invalid protocol
26013	08816	dharma/SQL JDBC errors	Invalid host name in connect string
26014	08809	dharma/SQL JDBC errors	Duplicate connection name
26015	08810	dharma/SQL JDBC errors	No active connection
26016	08812	dharma/SQL JDBC errors	Multiple local connections
26017	08813	dharma/SQL JDBC errors	Invalid protocol in connect_string
26018	08814	dharma/SQL JDBC errors	Exceeding permissible number of connections
26019	22818	dharma/SQL JDBC errors	Invalid date value
26020	22819	dharma/SQL JDBC errors	Invalid date string
26021	22820	dharma/SQL JDBC errors	Invalid number strings
26022	22821	dharma/SQL JDBC errors	Invalid number string
26023	22822	dharma/SQL JDBC errors	Invalid time value
26024	22523	dharma/SQL JDBC errors	Invalid time string
26025	22007	dharma/SQL JDBC errors	Invalid time stamp string

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
26026	s0501	dharma/SQL JDBC errors	SQL internal error
26027	s0502	dharma/SQL JDBC errors	Memory allocation failure
26028	22509	dharma/SQL JDBC errors	Inconsistent types
26029	22515	dharma/SQL JDBC errors	Missing input parameters
26030	0a000	dharma/SQL JDBC errors	Sorry, operation not yet implemented
26031	24538	dharma/SQL JDBC errors	Executing select statement
26032	24539	dharma/SQL JDBC errors	Cursor not closed
26033	24540	dharma/SQL JDBC errors	Open for non select statement
26034	22553	dharma/SQL JDBC errors	Overflow/Underflow error
26035	08555	dharma/SQL JDBC errors	Database not specified or improperly specified
26036	08556	dharma/SQL JDBC errors	Database not specified or Database not started
26037	22562	dharma/SQL JDBC errors	Input parameter size too high
26038	26000	dharma/SQL JDBC errors	Invalid statement type
26039	07001	dharma/SQL JDBC errors	Wrong number of parameters
26040	2250x	dharma/SQL JDBC errors	Numeric value out of range.
26041	2250y	dharma/SQL JDBC errors	Data truncated.
26042	22800	dharma/SQL JDBC errors	Remote procedure call error
26043	S1001	dharma/SQL JDBC errors	Memory allocation failure
26044	S1109	dharma/SQL JDBC errors	Invalid Cursor position
26045	S1010	dharma/SQL JDBC errors	Function sequence error
26046	24000	dharma/SQL JDBC errors	Invalid cursor state
26047	S1106	dharma/SQL JDBC errors	Fetch type out of range
26048	S1107	dharma/SQL JDBC errors	Row value out of range
26049	S1002	dharma/SQL JDBC errors	Invalid column number
26050	S1C00	dharma/SQL JDBC errors	Driver not capable
26051	S1T00	dharma/SQL JDBC errors	Timeout expired
26052	S1090	dharma/SQL JDBC errors	Invalid string or buffer length
26053	34000	dharma/SQL JDBC errors	Invalid cursor name
26054	3C000	dharma/SQL JDBC errors	Duplicate cursor name
26055	S1009	dharma/SQL JDBC errors	Invalid argument value

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
26056	S1004	dharma/SQL JDBC errors	SQL data type out of range
26057	S1008	dharma/SQL JDBC errors	Operation canceled
26058	S1094	dharma/SQL JDBC errors	Invalid scale value
26059	S1104	dharma/SQL JDBC errors	Invalid precision value
26060	S1093	dharma/SQL JDBC errors	Invalid parameter number
26061	S1009	dharma/SQL JDBC errors	Invalid argument value
26062	08004	dharma/SQL JDBC errors	Data source rejected establishment of connection
26063	S1000	dharma/SQL JDBC errors	General error
26064	01002	dharma/SQL JDBC errors	Disconnect error
26065	IM001	dharma/SQL JDBC errors	Driver does not support this function
26066	IM002	dharma/SQL JDBC errors	Data source not found and no default driver specified
26067	IM003	dharma/SQL JDBC errors	Specified driver could not be loaded
26068	S1015	dharma/SQL JDBC errors	No cursor name available
26069	07006	dharma/SQL JDBC errors	Restricted data type attribute violation
26070	01000	General Warning	“One or more of the returned String fields have been truncated."
26071	01000	General Warning	“One or more null values were ignored in the computation of an aggregate function such as SUM, AVG, MIN, and MAX.”
26072	01000	General Warning	"Number of items in the SELECT list does not equal the number of host variables in the INTO clause"
26073	01000	General Warning	"UPDATE or DELETE statement did not have a WHERE clause.”
26074	01000	General Warning	"SQL Kernel had to perform an implicit rollback due to a system failure and/or due to a deadlock situation"
30001	5050w	dharma/SQL rds errors	Query aborted on user request
30002	k0k02	dharma/SQL network interface	invalid network handle
30003	k0k03	dharma/SQL network interface	invalid sqlnetwork INTERFACE
30004	k0k04	dharma/SQL network interface	invalid sqlnetwork INTERFACE procedure
30005	k0k05	dharma/SQL network interface	INTERFACE is already attached

Table B-1: Error Codes and Messages

ErrorCode	SQLSTATE Value	Class Condition	Subclass Message
30006	k0k06	dharma/SQL network interface	INTERFACE entry not found
30007	k0k07	dharma/SQL network interface	INTERFACE is already registered
30008	k0k08	dharma/SQL network interface	mismatch in pkt header size and total argument size
30009	k0k09	dharma/SQL network interface	invalid server id
30010	k0k10	dharma/SQL network interface	reply does not match the request
30011	k0k02	dharma/SQL network interface	memory allocation failure
30031	k0k11	dharma/SQL network interface	error in transmission of packet
30032	k0k12	dharma/SQL network interface	error in reception of packet
30033	k0k13	dharma/SQL network interface	no packet received
30034	k0k14	dharma/SQL network interface	connection reset
30051	k0k15	dharma/SQL network interface	network handle is inprocess handle
30061	k0k16	dharma/SQL network interface	could not connect to sql network daemon
30062	k0k17	dharma/SQL network interface	error in number of arguments
30063	k0k18	dharma/SQL network interface	requested INTERFACE not registered
30064	k0k19	dharma/SQL network interface	invalid INTERFACE procedure id
30065	k0k20	dharma/SQL network interface	requested server executable not found
30066	k0k21	dharma/SQL network interface	invalid configuration information
30067	k0k22	dharma/SQL network interface	INTERFACE not supported
30091	k0k23	dharma/SQL network interface	invalid service name
30092	k0k24	dharma/SQL network interface	invalid host
30093	k0k25	dharma/SQL network interface	error in tcp/ip accept call
30094	k0k26	dharma/SQL network interface	error in tcp/ip connect call
30095	k0k27	dharma/SQL network interface	error in tcp/ip bind call
30096	k0k28	dharma/SQL network interface	error in creating socket
30097	k0k29	dharma/SQL network interface	error in setting socket option
30101	k0k30	dharma/SQL network interface	interrupt occurred
30102	k0k31	dharma/SQL network interface	Client/Server not WideChar Compatible
40001	L0L01	dharma/SQL env error	Error in reading configuration
50000	60614	dharma/SQL DHRSS errors	DHRSS- Improper call to DFLT SS

Appendix C

System Limits

C.1 MAXIMUM VALUES

The following table lists the maximum sizes for various attributes of the Dharma SDK Server environment.

Table C-1: Dharma SDK Server System Limits

Attribute	Name	Value
Maximum number of procedure arguments in an SQL CALL statement	TPE_MAX_PROC_ARGS	50
Maximum length of an SQL statement	TPE_MAX_SQLSTMTLEN	35000
Maximum length of a column: standard data types	TPE_MAX_FLDLEN	2000
Maximum length of a column: VARBINARY and VARCHAR specifying the character set designated as NATIONAL CHARACTER by the underlying storage system	TPE_EXT_MAX_FLDLEN	32752
Maximum length of default value specification	TPE_MAX_DFLT_LEN	250
Maximum length for an identifier (	TPE_MAX_IDLEN	32
Maximum length of a connect string	TPE_MAX_CONNLEN	100
Maximum length for a user-name in a connect string	TPE_UNAME_LEN	32
Maximum number of database connections	TPE_MAX_NO_CONN	10
Maximum length of an error message	TPE_MAX_ERRLEN	256
Maximum number of columns in a table	TPE_MAX_FIELDS	500
Maximum number of index components for a table, for all indexes on that table	SQL_MAXIDXENTRIES	100
Maximum length of a CHECK constraint clause	SQL_MAXCHKCL_SZ	240
Maximum number of check constraints in a table	SQL_MAXCHKCNSTRS	1000
Maximum number of foreign constraints in a table	SQL_MAXFRNCNSTRS	1000
Maximum number of nesting levels in an SQL statement	SQL_MAXLEVELS	25
Maximum number of table references in an SQL statement: Microsoft Windows	SQL_MAXTBLREF	50
Maximum number of table references in an SQL statement: other platforms	SQL_MAXTBLREF	250
Maximum size of input parameters for an SQL statement	SQL_MAXIPARAMS_SZ	512

Table C-1: Dharma SDK Server System Limits

Attribute	Name	Value
Maximum number of outer references in an SQL statement	SQL_MAX_OUTER_REF	25
Maximum nesting level for view references	MAX_VIEW_LEVEL	25
Specifies the size, in kilobytes, of the main memory cache used by the flat file system for temporary tables. The default value is 1000-KB of memory. The main-memory storage sys-tem uses the cache for storing temporary tables for sorting and creating dynamic indexes during processing. If you have more than 8-MB of memory, increasing TPE_MM_CACHESIZE should improve performance by reducing the need to write to the main-memory storage system's swap file.	TPE_MM_CACHESIZE	1000
Limits the maximum size of a row the main-memory storage system uses in internal temporary tables.Increase this value if complex queries generate result sets whose rows are greater than 4 KB. Com-plex queries include those that join multiple tables or include the ORDER BY clause in SELECT statements.	TPE_MM_BLOCKSIZE	4
Specifies the maximum size, in kilobytes, of the swap file the main-memory storage system uses when it writes to disk from its main memory cache. The default size is 500,000-KB. This limits the total data size to be sorted for a given query. Sorting of data more than this size will result in error (-16001): MM- No data block. Depending on the data size, this need to be changed.The swap file is created in the directory specified by the TPE_DATADIR runtime vari-able.	TPE_MM_SWAPSIZE	500,000
If set to TRUE, directs the main-memory storage system to use a system-provided library quick sort procedure, instead of the built-in quick sort.	TPE_MM_USE_SYS_QSORT	

Appendix D

Glossary

D.1 TERMS

add [an ODBC data source]
Make a data source available to ODBC through the Add operation of the ODBC Administrator utility. Adding a data source tells ODBC where a specific database resides and which ODBC driver to use to access it. Adding a data source also invokes a setup dialog box for the particular driver so you can provide other details the driver needs to connect to the database.

cardinality
Number of rows in a result table.

client
Generally, in client/server systems, the part of the system that sends requests to servers and processes the results of those requests.

client/server configuration
The version of the Dharma SDK that implements a network ODBC architecture. In the client/server configuration, the ODBC tool and the Dharma SDK Driver runs on Windows, while the Dharma SDK Server runs on the UNIX or NT server hosting the proprietary storage system.

data dictionary
Another term for system catalog.

data source
See ODBC data source

desktop configuration
The version of the Dharma SDK that implements a "single-tier" ODBC architecture. In the desktop configuration, the ODBC tool, the Dharma SDK software, and the proprietary data all reside on the same Windows computer.

dharma
The default owner name for all system tables in a Dharma database. Users must qualify references to system tables as dharma.table-name.

metadata
Data that details the structure of tables and indexes in the proprietary storage system. The SQL engine stores metadata in the system catalog.

ODBC application
Any program that calls ODBC functions and uses them to issue SQL statements. Many vendors have added ODBC capabilities to their existing Windows-based tools.

ODBC data source
In ODBC terminology, a specific combination of a database system, the operating system it uses, and any network software required to access it. Before applications can access a database through ODBC, you use the ODBC Administrator to add a data source -- register information about the database and an ODBC driver that can connect to it -- for that database. More than one data source name can refer to the same database, and deleting a data source does not delete the associated database.

ODBC driver
Vendor-supplied software that processes ODBC function calls for a specific data source. The driver connects to the data source, translates the standard SQL statements into syntax the data source can process, and returns data to the application. The Dharma SDK includes an ODBC driver that provides access to proprietary storage systems underlying the ODBC server.

ODBC driver manager
A Microsoft-supplied program that routes calls from an application to the appropriate ODBC driver for a data source.

ODBC server
The executable that results from building an implementation of the storage interfaces with the SQL engine library. To get started with the ODBC SDK, you can build an ODBC Server from the supplied sample implementation of the storage interfaces. Eventually, you will build an ODBC Server from your own implementation of the storage system to provide access to a proprietary storage system.

primary key
A subset of the fields in a table, characterized by the constraint that no two records in a table may have the same primary key value, and that no fields of the primary key may have a null value. Primary keys are specified in a CREATE TABLE statement.

query expression
The fundamental element in SQL syntax. Query expressions specify a result table derived from some combination of rows from the tables or views identified in the FROM clause of the expression. Query expressions are the basis of SELECT, CREATE VIEW, and INSERT statements

result set
Another term for result table.

result table
A temporary table of values derived from columns and rows of one or more tables that meet conditions specified by a query expression.

row identifier
Another term for tuple identifier.

search condition
The SQL syntax element that specifies a condition that is true or false about a given row or group of rows. Query expressions and UPDATE statements can specify a search condition. The search condition restricts the number of rows in the result table for the query expression or UPDATE statement. Search conditions contain one or more predicates. Search conditions follow the WHERE or HAVING keywords in SQL statements.

selectivity
The fraction of a table's rows returned by a query.

server
Generally, in client/server systems, the part of the system that receives requests from clients and responds with results to those requests.

SQL engine
The core component of the Dharma SDK Server. The SQL engine receives requests from the Dharma SDK Driver, processes them, and returns results.

storage interfaces
C routines called by the SQL engine that access and manipulate data in a proprietary storage system. A proprietary storage system must implement supplied storage stub templates to map the storage interfaces to the underlying storage system.

storage manager
A completed implementation of the Dharma SDK storage interfaces. A storage manager receives calls from the SQL engine and accesses the underlying proprietary storage system to retrieve and store data.

storage interfaces
Another term for stub interfaces.

storage system
The proprietary database system that underlies a storage manger. The Dharma SDK provides a SQL interface to a storage system through the SQL engine and its stub interfaces.

stub interfaces
Template routines provided with the Dharma SDK for implementing access to proprietary storage systems. Once filled in for a particular storage system, the completed stubs are called storage managers.

stubs
Another term for stub interfaces.

system catalog
Tables created by the SQL engine that store information about tables, columns, and indexes that make up the database. The SQL engine creates and manages the system catalog independent of the proprietary storage system.

system tables
Another term for system catalog.

tid
Another term for tuple identifier.

transaction
A group of database operations whose changes can be made permanent or undone only as a unit.

tuple identifier
A unique identifier for a tuple (row) in a table. Storage managers return a tuple identifier for the tuple that was inserted after an insert operation. The SQL engine passes a tuple identifier to the delete, update, and fetch stubs to indicate which tuple is affected.

The SQL scalar function ROWID and related functions return tuple identifiers to applications.

view
A virtual table that recreates the result table specified by a SELECT statement. No data is stored in a view, but other queries can refer to it as if it were a table containing data corresponding to the result table it specifies.

Index

F

T

U

W

Y