

Daffodil DB

SQL Reference Guide

Version 4.1

March 2005

Copyright © Daffodil Software Limited
Sco 42, 3rd Floor
Old Judicial Complex, Civil Lines
Gurgaon - 122001
Haryana, India.
www.daffodildb.com

All rights reserved. Daffodil DB™ is a registered trademark of Daffodil Software Limited. Java™ is a registered trademark of Sun Microsystems, Inc. All other brand and product names are trademarks of their respective companies.

Table of Contents

1 Preface.....	7
1.1 Purpose of Document.....	7
1.2 Audience.....	7
2 Conventions.....	8
3 Related Document	9
4 Keywords.....	10
3.1 Reserved Words.....	10
3.2 Non-Reserved Keywords...11	
5 Identifier.....	13
5.1 Regular Identifier.....13	
5.2 Delimited Identifier.....14	
6 Data Types.....	15
6.1 Predefined Type.....15	
6.1.1 Character String Type..... 15	
6.1.1.1 Character <i>or</i> Char.....16	
6.1.1.2 Character Varying <i>or</i> Char Varying <i>or</i> Varchar <i>or</i> Varchar2.....17	
6.1.1.3 Character Large Object <i>or</i> Char Large Object <i>or</i> CLOB <i>or</i> Long 17	
6.1.2 Binary Large Object String Type....19	
6.1.2.1 Binary.....19	
6.1.2.2 Varbinary.....20	
6.1.2.3 BLOB <i>or</i> Long Varbinary.....20	
6.1.3 Numeric Type.....21	
6.1.3.1 Exact Numeric Type.....21	
6.1.3.1.1 NUMERIC <i>or</i> DECIMAL <i>or</i> DEC <i>or</i> NUMBER.....22	
6.1.3.1.2 INTEGER <i>or</i> INT.....23	
6.1.3.1.3 SMALLINT.....23	
6.1.3.1.4 LONG <i>or</i> BIGINT.....24	
6.1.3.1.5 BYTE <i>or</i> TINTINT....25	
6.1.3.2 Approximate Numeric Type.....25	
6.1.3.2.1 Float.....26	
6.1.3.2.2 Real.....27	
6.1.3.2.3 Double Precision.....27	
6.1.4 Boolean.....28	
6.1.5 Date time Type.....28	
6.1.5.1 Date.....29	
6.1.5.2 Time.....30	
6.1.5.3 Time Stamp.....31	
6.2 Domain Name.....31	
7 Literals.....	32
7.1 Character String Literal.....32	
7.2 Numeric Literal.....32	
7.3 Date Time Literal.....33	
7.4 Boolean Literal.....35	
8 Functions	36
8.1 Numeric Functions.....36	
8.1.1 Absolute Value Expression	

8.1.2	Modulus Value Expression	
8.1.3	Sine Function	
8.1.4	Power Function	
8.1.5	Rand Function	
8.1.6	SQRT Function	
8.1.7	TRUNCATE Function	
8.1.8	FLOOR Function	
8.1.9	CEILING Function	
8.1.10	LOG Function	
8.1.11	EXP Function	
8.1.12	COS Function	
8.1.13	TAN Function	
8.1.14	COT Function	
8.1.15	ACOS Function	
8.1.16	ASIN Function	
8.1.17	ATAN Function	
8.1.18	DEGREES Function	
8.1.19	RADIANS Function	
8.1.20	PI Function	
8.1.21	ATAN2 Function	
8.1.22	ROUND Function	
8.1.23	SIGN Function	
8.2	Date Time Functions.....	52
8.2.1	DAYNAME Function	
8.2.2	DAYOFMONTH Function	
8.2.3	DAYOFWEEK Function	
8.2.4	DAYOFYEAR Function	
8.2.5	WEEK Function	
8.2.6	MONTH Function	
8.2.7	YEAR Function	
8.2.8	MONTHNAME Function	
8.2.9	HOUR Function	
8.2.10	MINUTE Function	
8.2.11	SECOND Function	
8.2.12	TIMESTAMPADD Function	
8.2.13	TIMESTAMPDIFF Function	
8.2.14	CURDATE Function	
8.2.15	CURTIME Function	
8.2.16	CURTIMESTAMP Function	
8.2.17	DATE Function	
8.2.18	TIME Function	
8.3	String Functions.....	64
8.3.1	ASCII Value Function	
8.3.2	Left Function	
8.3.3	Right Function	
8.3.4	Space Function	
8.3.5	Replace Function	
8.3.6	Repeat Function	

8.3.7	Soundex Function	
8.3.8	Insert Function	
8.3.9	Difference Function	
8.3.10	Concat Function	
8.3.11	Locate Function	
8.3.12	Lcase Function	
8.3.13	Ucase Function	
8.3.14	Ltrim Function	
8.3.15	Rtrim Function	
8.3.16	Char Function	
8.3.17	Length Function	
8.3.18	Substring Function	
8.3.19	EqualsCaseSensitive Function	
8.4	System Functions.....	81
8.4.1	Current Database Function or CURRENT_DATABASE	
8.4.2	User Function or CURRENT_USER	
8.4.3	IFNULL Function	
8.5	Special Functions.....	82
7.5.1	TOP Function	
8.6	Aggregate Functions.....	83
7.6.1	Count	
7.6.2	Avg	
7.6.3	Sum	
7.6.4	Max/Min	
9	Expressions.....	85
9.1	Numeric Expression.....	85
9.2	Boolean Expression.....	87
9.3	String Expression.....	89
9.4	Expression Primary.....	91
9.4.1	SubQuery	
9.4.2	COLUMN REFERENCE	
9.4.3	CONSTANT	
9.4.4	MULTI-VALUED EXPRESSION	
9.4.5	PARENTHESESIZED EXPRESSION	
10	Predicates	95
10.1	Comparison Predicate.....	96
10.2	Between Predicate.....	98
10.3	Like Predicate.....	100
10.4	Exists Predicate.....	101
10.5	In Predicate.....	102
10.6	Null Predicate.....	103
10.7	Quantified Comparison Predicate...	104
10.8	Contains Predicate*	107
11	Data Definition Language.....	110
11.1	Create Table Statement.....	112
11.2	Create Trigger Statement.....	121
11.3	Create Procedure Statement.....	124
11.4	Create View Statement.....	135

11.5 Create Index Statement.....	136
11.6 CreateFullTextIndex*.....	137
11.7 Create Domain Statement.....	139
11.8 Create Schema Statement	139
11.9 Create User Statement.....	140
11.10 Alter Table Statement.....	141
11.11 Drop Table Statement.....	143
11.12 Drop View Statement.....	144
11.13 Drop Index Statement.....	144
11.14 DropFullTextIndex*.....	144
11.15 Drop Schema Statement.....	144
11.16 Drop Procedure Statement.....	145
11.17 Drop Trigger Statement.....	146
12 Persistent Stored Modules	147
13 Data Manipulation Language.....	149
13.1 Insert Statement.....	149
13.2 Update Statement.....	151
13.3 Delete Statement.....	152
14 Data Query And Control Language	153
14.1 Select Statement.....	153
14.1.1 FROM Clause	
14.1.2 JOIN Operators	
14.1.3 GROUP BY Clause	
14.1.4 UNION/INTERSECTOR Operator	
14.1.5 ORDER BY Clause	
14.1.6 Alias Support	
14.1.7 Comments Support	
15 Call Statement.....	171
16 Session And Transaction Control Statements.....	172
16.1 Set Transaction Statement	172
16.2 Savepoint Statement.....	172
16.3 Commit Statement.....	172
16.4 Rollback Statement.....	173
16.5 Set Session Authrization.....	173
16.6 Set Session Characteristics Statement.....	174
17 SQL Security And Privileges.....	175
17.1 Grant Statement.....	176
17.2 Revoke Statement	178
17.3 Create Role	
17.4 Grant Role	
17.5 Drop Role	
18 Appendix.....	183
18.1 Error Messages	
18.2 Country Codes	
18.3 Language Codes	

* Features that are not supported in One\$DB
--

Preface

Purpose of This Document

SQL Reference Guide is a comprehensive document which covers all the SQL-99 features supported by Daffodil DB. This ready reference tool describes in detail the syntax and semantics of SQL language statements and elements for Daffodil DB. It explains how to use SQL with Daffodil DB and how to perform various database operations on Daffodil DB such as creating tables or indexes, managing transactions and sessions, Daffodil DB security features etc.

Audience

This guide is intended to act as a ready reference tool for software developers building Daffodil DB applications. This guide assumes that that you are familiar with the following concepts:

- Basic SQL (Structured Query Language).
- Basic Database Concepts.
- Basic Java Programming Language.

It is also assumed that the reader has gone through [“Getting Started with Daffodil DB Guide”](#).

Conventions

This section describes documentation syntax conventions. *Syntax* conventions convey specific information on keywords and clauses in the SQL statements described in the document.

Syntax Conventions

Each SQL statement uses certain types of capitalization, formatting, and punctuation that describe the attributes of different portions of the statement.

- UPPERCASE If a portion of an SQL statement occurs in UPPERCASE, then the capitalized words are keywords, which are usually required in the SQL statement or clause. Keywords are not case sensitive, and they must be spelled exactly the way they display in the document.
- [] Clauses in an SQL statement that occur between [brackets] are optional. If an optional clause has several components or keywords, they occur within brackets.
- { } Curly 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.
- ... Ellipses in an SQL statement have similar meaning as “etc.” It denotes that a series of keywords, clauses, or variables that precede the ellipses can go on indefinitely.
- | Vertical bars in SQL statement separate a set of options.
- (), ; Parentheses and other punctuation marks are required elements. Enter them as shown in syntax diagrams.

Related Documentation

Daffodil DB Getting Started Guide	Designed to help new and intermediate Daffodil DB users navigate and perform common tasks like How to start and stop Daffodil DB, Understanding key variables used by Daffodil DB, User documentation bundled with Daffodil DB. Also briefly describes Daffodil DB Editions and Tools
Daffodil DB System Guide	Describes the architecture of Daffodil DB and provides the information that the server administrator might need to keep Daffodil DB running with high performance and reliability in a server framework or a multi-user application server. Also describes the standards on which Daffodil DB had been built, transaction capabilities and some of the unique features supported by Daffodil DB.
Daffodil DB JDBC Reference Guide	Explains how to use Daffodil DB and JDBC technology to develop applications. It describes the basic Daffodil DB and JDBC concepts like JDBC 3.0 features supported by Daffodil DB, how to create and access Daffodil DB databases through JDBC API, Daffodil DB support for JDBC and JTA and how to use Daffodil DB in a Distributed Transaction Processing environment.
Daffodil DB Tools Guide	Explains how to use Daffodil DB Browser with Embedded as well as Server versions of Daffodil DB. Describes how to perform various database operations on Daffodil DB using Daffodil DB Browser such as creating a database, creating database objects, manipulating data, creating triggers etc.

Keywords

Reserved Words

Daffodil DB reserves certain keywords as Reserved Words which cannot be used, as an identifier for a table, column, or index, or as a correlation name defined in a SELECT statement, unless you delimit them. A delimited identifier is an identifier specified in double quotes. Any word, including keywords, can be a delimited identifier.

Daffodil DB Reserved Words

ABSOLUTE	DEFERRED	LEFT	ROLLUP
ACTION	DELETE	LESS	ROUTINE
ADD	DEPTH	LEVEL	ROW
ADMIN	DEREF	LIKE	ROWNUM
AFTER	DESC	LIMIT	ROWS
AGGREGATE	DESCRIBE	LOCAL	SAVEPOINT
ALIAS	DESCRIPTOR	LOCALTIME	SCHEMA
ALL	DESTROY	LOCALTIMESTAMP	SCROLL
ALLOCATE	DESTRUCTOR	LOCATOR	SCOPE
ALTER	DETERMINISTIC	LONG	SEARCH
AND	DICTIONARY	MIN	SECOND
ANY	DIAGNOSTICS	MAP	SECTION
ARE	DISCONNECT	MAX	SELECT
ARRAY	DISTINCT	MATCH	SEQUENCE
AS	DOMAIN	MINUTE	SESSION
ASC	DOUBLE	MODIFIES	SESSION_USER
ASSERTION	DROP	MODIFY	SET
AT	DYNAMIC	MODULE	SETS
AUTHORIZATION	EACH	MONTH	SIZE
AVG	ELSE	NAMES	SMALLINT
BEFORE	END	NATIONAL	SOME
BEGIN	EQUALS	NATURAL	SPACE
BIGINT	ESCAPE	NCHAR	SPECIFIC
BINARY	EVERY	NCLOB	SPECIFICTYPE
BIT	EXCEPT	NEW	SQL
BLOB	EXCEPTION	NEXT	SQLException
BOOLEAN	EXEC	NO	SQLSTATE
BOTH	EXECUTE	NONE	SQLWARNING
BREADTH	EXTERNAL	NOT	START
BY	FALSE	NULL	STATEMENT
BYTE	FETCH	NUMERIC	STATIC
CALL	FIRST	OBJECT	STRUCTURE
CASCADE	FLOAT	OF	SUM
CASCADED	FOR	OFF	SYSTEM_USER
CASE	FOREIGN	OLD	TABLE
CAST	FOUND	ON	TEMPORARY
CATALOG	FROM	ONLY	TERMINATE
CHAR	FREE	OPEN	THAN
CHARACTER	FULL	OPERATION	THEN
CHECK	FUNCTION	OR	TIME
CLASS	GENERAL	ORDER	TIMESTAMP
CLOB	GET	ORDINALITY	TIMEZONE_HOUR
CLOSE	GLOBAL	OUT	TIMEZONE_MINUTE
COLLATE	GO	OUTER	TINYINT
COLLATION	GOTO	OUTPUT	TO
COLUMN	GRANT	PAD	TRAILING
COMMIT	GROUP	PARAMETER	TRANSACTION
COMPLETION	GROUPING	PARTIAL	TRANSLATION
CONNECT	HAVING	PATH	TREAT
CONNECTION	HOST	POSTFIX	TRIGGER
CONSTRAINT	HOURL	PRECISION	TRUE
CONSTRAINTS	IDENTITY	PREFIX	UNDER
CONSTRUCTOR	IGNORE	PREORDER	UNION

CONTINUE	IMMEDIATE	PREPARE	UNIQUE
CORRESPONDING	IN	PRESERVE	UNKNOWN
COUNT	INDICATOR	PRIMARY	UNNEST
CREATE	INITIALIZE	PRIOR	UPDATE
CROSS	INITIALLY	PRIVILEGES	USAGE
CUBE	INNER	PROCEDURE	USER
CURRENT	INOUT	PUBLIC	USING
CURRENT_DATE	INPUT	READ	VALUE
CURRENT_PATH	INSERT	READS	VALUES
CURRENT_ROLE	INT	REAL	VARBINARY
CURRENT_TIME	INTEGER	RECURSIVE	VARCHAR
CURRENT_TIMESTAMP	INTERSECT	REF	VARIABLE
CURRENT_USER	INTERVAL	REFERENCES	VARYING
CURSOR	INTO	REFERENCING	VIEW
CYCLE	IS	RELATIVE	WHEN
DATA	ISOLATION	RELEASE	WHENEVER
DATE	ITERATE	RESTRICT	WHERE
DAY	JOIN	RESULT	WITH
DEALLOCATE	KEY	RETURN	WITHOUT
DEC	LANGUAGE	RETURNS	WORK
DECIMAL	LARGE	REVOKE	WRITE
DECLARE	LAST	RIGHT	YEAR
DEFAULT	LATERAL	ROLE	ZONE
DEFERRABLE	LEADING	ROLLBACK	

NOTE: Words listed here are SQL reserved words and should not be used. Some of these keywords may not be supported in the current version, but are reserved for future versions of Daffodil DB.

Non-Reserved Words

Daffodil DB reserves certain words as Non Reserved Words. Non-Reserved words can be used as an identifier for a table, column, or index, or as a correlation name, which is defined in a SELECT statement.

Daffodil DB Non-Reserved Words

ABS	EXISTING	REPEAT
ACOS	EXISTS	REPLACE
ADA	EXP	RETURNED_LENGTH
ASENSITIVE	EXTRACT	RETURNED_OCTET_LENGTH
ASCII	FINAL	RETURNED_SQLSTATE
ASSIGNMENT	FLOOR	ROUND
ASENSITIVE	FORTRAN	ROUTINE_CATALOG
ASIN	G	ROUTINE_NAME
ASYMMETRIC	GENERATED	ROUTINE_SCHEMA
ATAN	GRANTED	ROW_COUNT
ATAN2	HIERARCHY	RTRIM
ATOMIC	HOLD	SCALE
ATTRIBUTE	IFNULL	SCHEMA_NAME
B	IMPLEMENTATION	SECURITY
BETWEEN	INDEX	SELF
BIT_LENGTH	INFIX	SENSITIVE
BITVAR	INSENSITIVE	SERIALIZABLE
C	INSTANCE	SERVER_NAME
CALLED	INSTANTIABLE	SIMPLE
CARDINALITY	INVOKER	SIGN
CATALOG_NAME	K	SIN

CEILING	KEY_MEMBER	SOUNDEX
CHAIN	KEY_TYPE	SOURCE
CHAR_LENGTH	LCASE	SPECIFIC_NAME
CHARACTER_LENGTH	LENGTH	SIMILAR
CHARACTER_SET_CATALOG	LOCATE	SQL_TSI_DAY
CHARACTER_SET_NAME	LOG	SQL_TSI_FRAC_SECOND
CHARACTERISTICS	LOWER	SQL_TSI_HOUR
CHARACTER_SET_SCHEMA	LTRIM	SQL_TSI_MINUTE
CHECKED	M	SQL_TSI_MONTH
CLASS_ORIGIN	MESSAGE_LENGTH	SQL_TSI_QUARTER
COALESCE	MESSAGE_OCTET_LENGTH	SQL_TSI_SECOND
COBOL	MESSAGE_TEXT	SQL_TSI_WEEK
COLLATION_CATALOG	METHOD	SQL_TSI_YEAR
COLLATION_NAME	MOD	SQRT
COLLATION_SCHEMA	MONTHNAME	SUBLIST
COLUMN_NAME	MORE	SUBSTRING
COMMAND_FUNCTION	MOUNT	STATE
COMMAND_FUNCTION_CODE	MUMPS	STYLE
COMMITTED	NAME	SUBCLASS_ORIGIN
CONCAT	NOW	SYMMETRIC
CONDITION_NUMBER	NULLABLE	SYSTEM
CONNECTION_NAME	NUMBER	TABLE_NAME
CONSTRAINT_CATALOG	NULLIF	TAN
CONSTRAINT_NAME	OCTET_LENGTH	TIMESTAMPADD
CONSTRAINT_SCHEMA	OPTIONS	TIMESTAMPDIFF
CONTAINS	ORDERING	TOP
CONVERT	OVERLAPS	TRANSACTIONS_COMMITTED
COS	OVERLAY	TRANSACTIONS_ROLLED_BACK
COT	OVERRIDING	TRANSACTION_ACTIVE
CURDATE	PASCAL	TRANSFORM
CURRENT_DATABASE	PARAMETER_MODE	TRANSFORMS
CURSOR_NAME	OPTION	TRANSLATE
CURTIME	PARAMETERS	TRIGGER_CATALOG
DATABASE	PARAMETER_NAME	TRIGGER_SCHEMA
DATETIME_INTERVAL_CODE	PARAMETER_ORDINAL_POSITION	TRIGGER_NAME
DATETIME_INTERVAL_PRECISION	PARAMETER_SPECIFIC_CATALOG	TRIM
DAYNAME	PARAMETER_SPECIFIC_NAME	TRUNCATE
DAYOFMONTH	PARAMETER_SPECIFIC_SCHEMA	TYPE
DAYOFWEEK	PASSWORD	UCASE
DAYOFYEAR	PI	UNCOMMITTED
DEFINED	PLI	UNNAMED
DEFINER	PLACING	UPPER
DEGREES	POSITION	USER_DEFINED_TYPE_CATALOG
DERIVED	POWER	USER_DEFINED_TYPE_NAME
DIFFERENCE	RADIANS	USER_DEFINED_TYPE_SCHEMA
DISPATCH	RAND	WEEK
DYNAMIC_FUNCTION	REPEATABLE	X
DYNAMIC_FUNCTION_CODE		

NOTE: Words listed here are SQL Non-Reserved words and can be used freely. Some of these Non-Reserved Words may not be supported in the current version, but are reserved for future versions of Daffodil DB.

Identifier

An SQL identifier can be a Regular Identifier or it can be a Delimited Identifier. Delimited Identifiers are enclosed in double quotes.

Syntax

<Identifier>::=

<regular identifier>

<delimited identifier>

Regular Identifier

An SQL-99 identifier is a dictionary object identifier that conforms to the rules of SQL-99. SQL-99 states that identifiers for dictionary objects are limited to 128 characters and are case-insensitive (unless delimited by double quotes). User cannot use reserved words as identifiers for dictionary objects unless they are delimited.

Syntax

<regular identifier> ::= <identifier start> [{ <underscore> <identifier part> } ...]

<identifier start> ::= <initial alphabetic character>

<underscore>

<identifier part> ::= <initial alphabetic character>

<digit>

<initial alphabetic character> ::= <simple latin lower case letter>

<simple latin upper case letter>

<simple latin lower case letter>::=

a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

<simple latin upper case letter>::=

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

<underscore> ::= _

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

Identifier Start

Identifier Start should be an alphabetic character and an underscore. It should not contain any digit or special characters.

Identifier Part

Identifier Part should be an alphabetic character, an underscore and a sequence of digits. It should not include any special character except underscore.

Examples of valid identifiers

ABCDEDF

_ABCDE

_A1234

Examples of invalid identifiers

1ABCD (starts with digit)

Abc\$% (includes special characters \$ %)

Delimited Identifier

A delimited identifier is an identifier specified in double quotes. Any word, including keywords, can be a delimited identifier. Enclosing a name in double quotation marks preserves the case of the name. A reserved word can be part of an identifier, such as DEFAULT_TABLE, only to the extent when it is not exactly the same as the keyword itself.

Syntax

<delimited identifier> ::= <double quote> <delimited identifier body>... <double quote>

<delimited identifier body> ::= <Simple Latin Letter>| <Special Characters>| <digit>

Simple Latin Letter

Simple Latin Letter is a collection of simple Latin upper case letter and a collection of simple Latin lower case letter. It means delimited identifier can contain any alphabetic character.

Special Characters

Special Characters is a collection of special character symbols like !@#%&*()-+=etc. and any digit (0-9). It means delimited identifier can contain any special character, as double quotes delimit the meaning of these characters in the identifier.

The enclosing quotation marks are not the part of an identifier; they indicate only its beginning and end. To include a double quotation mark character in a delimited identifier, precede it with another double quotation mark.

Example

Valid Delimited Identifiers are:

“SELECT” (double quotes delimit the meaning of reserved word “Select”)

“\$1234” (double quotes delimit the meaning of special character “\$”)

Data Types

Data type defines the type of data a column can contain.

Syntax

```
<data type> ::=  
<predefined type>  
| <domain name>
```

Predefined Type

These are the data types defined by Daffodil DB. They are:

- Character String
- Binary Large Object String
- Numeric
- Boolean
- Date-time

Syntax

```
<predefined type> ::=  
<character string type>  
| <binary large object string type>  
| <numeric type>  
| Boolean  
| <date-time type>
```

Character String Type

It stores character (alphanumeric) data, which are words and free-form text, in the database character set.

Syntax

```
<character string type> ::=  
CHARACTER [ <left paren> <length> <right paren> ]  
| CHAR [ <left paren> <length> <right paren> ]  
| CHARACTER VARYING <left paren> <length> <right paren>  
| CHAR VARYING <left paren> <length> <right paren>  
| VARCHAR <left paren> <length> <right paren>  
| VARCHAR
```

| VARCHAR2 [<left paren> <length> <right paren>]

| CHARACTER LARGE OBJECT [<left paren> <large object length> <right paren>]

| CHAR LARGE OBJECT [<left paren> <large object length> <right paren>]

| CLOB [<left paren> <large object length> <right paren>]

| LONG VARCHAR [<left paren> <large object length> <right paren>]

Character or Char

The **CHARACTER** or **char** data type accepts character strings of a fixed length. Length of the character string should be specified in the data type declaration.

Syntax	character[(n)], char[(n)]
Corresponding Compile-Time Java Type	<i>java.lang.String</i>
JDBC Metadata Type (java.sql.Types)	CHAR
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	N.A
Maximum Size	4 K

Note:- 1). (n) OR (ln) means length in digits.
 2). [(n)] OR [(ln)] means length in digits but optional.
 3.) [(p[,s))]] means precision and scale in digits but optional.

Example

CHARACTER (n) or Char (n)

where, n represents the desired length of the character string. The length parameter may take any value from 1 to 4192. If length is not specified during the declaration, then 1 is taken by default.

Character Varying *or* Char Varying *or* Varchar *or* Varchar2

This data type accepts character strings, of a variable length, up to the maximum length specified in the data type declaration. This declaration must include a positive integer in parentheses to define the maximum allowable character string length.

Syntax	character varying(<i>n</i>), char varying(<i>n</i>), varchar[(<i>n</i>)]
Corresponding Compile-Time Java Type	<i>java.lang.String</i>
JDBC Metadata Type (java.sql.Types)	VARCHAR
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	N.A
Maximum Size	4 K

Example

VARCHAR (*n*) or VARCHAR2 (*n*) or CHAR VARYING (*n*) or CHARACTER VARYING (*n*)
or VARCHAR or VARCHAR2

It can accept any length of character string up to *n* characters in length. The length parameter may take any value from 1 to 4192.

Character Large Object *or* Char Large Object *or* CLOB *or* Long Varchar

These data type accepts character strings, of a large length, up to the maximum length specified in the data type declaration. These are mostly used for columns, which can have data more than 4192 characters.

Syntax	character large object(<i>ln</i>), char large object(<i>ln</i>), clob[(<i>ln</i>)]
Corresponding Compile-Time Java Type	<i>java.sql.Clob</i>
JDBC Metadata Type (java.sql.Types)	CLOB
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	N.A

Example

CLOB (*n*) or CHAR LARGE OBJECT (*n*) or CHARACTER LARGE OBJECT (*n*) or LONG VARCHAR (*n*)

It can accept any length of character string up to *n* characters in length. The length parameter may take any value from 1 to 1073741823. If large object length is not specified then 1073741823, is taken by default.

Binary Large Object String Type

Syntax

<binary large object string type> ::=

BLOB [<left paren> <large object length> <right paren>]

| LONG VARBINARY [<left paren> <large object length> <right paren>]

| VARBINARY <left paren> <length> <right paren>

| VARBINARY

| BINARY [<left paren> <length> <right paren>]

Binary

The BINARY data type accepts binary strings, of a fixed length. Length of binary string should be specified in the data type declaration.

Syntax	binary[(n)]
Corresponding Compile-Time Java Type	java.lang.byte[]
JDBC Metadata Type (java.sql.Types)	BINARY
Default Value	Null
Minimum Value	-128
Maximum Value	127
Size	N.A
Maximum Size	4 K

BINARY (n)

Where, n represents desired length of the binary string. The length parameter may take any value from 1 to 4192. If no length is specified during the declaration, 1 is taken as default length.

Varbinary

VARBINARY data type accepts binary strings of a variable length, which is up to the maximum length specified in the data type declaration. These declarations may include a positive integer in the parentheses to define maximum allowable character string length.

Syntax	varbinary[(n)]
Corresponding Compile-Time Java Type	java.lang.byte[]
JDBC Metadata Type (java.sql.Types)	VARBINARY
Default Value	Null
Minimum Value	-128
Maximum Value	127
Size	N.A
Maximum Size	4 K

Example

VARBINARY (n) or VARBINARY

It can accept any length of binary string up to n characters in length. The length parameter may take any value from 1 to 4192. If no length is specified during the declaration, the default length is 1.

BLOB or Long Varbinary

These data type accepts binary strings, of a large length, and is up to the maximum length specified in the data type declaration. These are mostly used for columns, which can have data more than 4192 characters.

Syntax	blob[(ln)] or long varbinary[(ln)]
Corresponding Compile-Time Java Type	java.lang.Blob
JDBC Metadata Type (java.sql.Types)	BLOB or LONG VARBINARY
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	N.A
Maximum Size	1 GB

Example

BLOB(*n*) or LONG VARBINAY(*n*)

It can accept any length of binary string up to *n* characters in length. The length parameter may take any value from length 1 to 1073741823. If large object length is not specified, then 1073741823 is taken by default.

Numeric Type

The types which stores number type values.

Syntax

<numeric type> ::=

<exact numeric type>

| <approximate numeric type>

Exact Numeric Type

Exact numeric types are data types that store data in the form of numbers.

Syntax

<exact numeric type> ::=

NUMERIC [<left paren> <precision> [<comma> <scale>] <right paren>]

| DECIMAL [<left paren> <precision> [<comma> <scale>] <right paren>]

| DEC [<left paren> <precision> [<comma> <scale>] <right paren>]

| NUMBER [<left paren> <precision> [<comma> <scale>] <right paren>]

| INTEGER

| INT

| SMALLINT

| LONG

| BYTE

| TINYINT

| BIGINT

NUMERIC or DECIMAL or DEC or NUMBER

The **DECIMAL**, **NUMERIC**, **NUMBER** or **DEC** data types accept fixed-precision decimal values, for which you may define a precision and a scale in the data type declaration. The precision is a positive integer that indicates the number of digits that the number will contain. The scale is a positive integer that indicates a number of these digits that will represent decimal places to the right of the decimal point.

Syntax	<code>numeric[(p[,s])]</code> or <code>decimal[(p[,s])]</code> or <code>dec[(p[,s])]</code>
Corresponding Compile-Time Java Type	<code>java.math.BigDecimal</code>
JDBC Metadata Type (java.sql.Types)	NUMERIC or DECIMAL
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	N.A
Maximum Size	Numeric(38), Decimal(28)

These data types can be declared in any one of three different ways as illustrated below.

Examples

DECIMAL – Precision defaults to 38, Scale defaults to 0

DECIMAL (p) – Scale defaults to 0

DECIMAL (p, s) – Precision and Scale are defined by the user

NUMERIC – Precision defaults to 38, Scale defaults to 0

NUMERIC (p) – Scale defaults to 0

NUMERIC (p, s) – Precision and Scale are defined by the user

DEC – Precision defaults to 38, Scale defaults to 0

DEC (p) – Scale defaults to 0

DEC (p, s) – Precision and Scale are defined by the user

NUMBER – Precision defaults to 38, Scale defaults to 0

NUMBER (p) – Scale defaults to 0

NUMBER (p, s) – Precision and Scale are defined by the user

In the above examples, *p* is an integer representing precision and *s* is an integer representing scale.

INTEGER or INT

The INTEGER or INT data type accepts a 64-bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^{-31} and $2^{31} - 1$ (i.e. -2147483648 to 2147483647).

Syntax	int, integer
Corresponding Compile-Time Java Type	java.lang.Integer
JDBC Metadata Type (java.sql.Types)	INTEGER
Default Value	Null
Minimum Value	-2147483648
Maximum Value	2147483647
Size	4
Maximum Size	N.A

Examples of INTEGER or INT Valid Values

-2147483648

-1025

0

2147483647

SMALLINT

The SMALLINT data type accepts a 16 bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^{-15} and $2^{15} - 1$ (i.e. -32768 to 32767).

Syntax	smallint
Corresponding Compile-Time Java Type	java.lang.Short
JDBC Metadata Type (java.sql.Types)	SMALLINT
Default Value	Null
Minimum Value	-32768
Maximum Value	32767
Size	2
Maximum Size	N.A

Examples of SMALLINT Valid Values

-32768

0

32767

LONG or BIGINT

The **LONG** or **BIGINT** data type can accept numeric values up to 8 bytes. It stores any integer value between the range of 9223372036854775807 and -9223372036857447808.

Syntax	long, bigint
Corresponding Compile-Time Java Type	java.lang.Long
JDBC Metadata Type (java.sql.Types)	BIGINT
Default Value	Null
Minimum Value	-9223372036854775808
Maximum Value	9223372036854775807
Size	8
Maximum Size	N.A

Examples of BIGINT or LONG Valid Values

-3372036857447808

-857447808

0

23372036854775807

BYTE or TINYINT

The **BYTE or TINYINT** data type accepts an 8-bit signed integer value with an implied scale of zero. It stores any integer value between the range 2^{-7} and $2^7 - 1$ (i.e. -128 to 127).

Syntax	tinyint, byte
Corresponding Compile-Time Java Type	java.lang.Byte
JDBC Metadata Type (java.sql.Types)	TINYINT
Default Value	Null
Minimum Value	-128
Maximum Value	127
Size	1
Maximum Size	N.A

Examples of BYTE or TINYINT Valid Values

-128

0

127

Approximate Numeric Type

Approximate numeric data types stores data in form of numbers, which represents approximate values.

Syntax

<approximate numeric type> ::=

FLOAT [<left paren> <precision> <right paren>]

| REAL

| DOUBLE PRECISION

Float

The **FLOAT** data type accepts a double precision floating point number value. If no precision is specified during the declaration, the default precision is 15.

Syntax	float[(n)]
Corresponding Compile-Time Java Type	java.lang.Double
JDBC Metadata Type (java.sql.Types)	FLOAT
Default Value	Null
Minimum Value	4.9E-324
Maximum Value	1.7976931348623157E308
Size	4
Maximum Size	15

Examples of FLOAT (7) Valid Values

1234567

1.2

123.45678

-1234567

-1.2

-123.4567

Real

The REAL data type accepts single-precision floating point number values.

Syntax	real
Corresponding Compile-Time Java Type	java.lang.Float
JDBC Metadata Type (java.sql.Types)	REAL
Default Value	Null
Minimum Value	1.40E-45
Maximum Value	3.4028235E38
Size	4
Maximum Size	N.A

Examples of REAL Valid Values

-2345

0

1E-3

1.245

123456789012345678901234567890

Double Precision

The **DOUBLE PRECISION** data type accepts a double precision floating point value. No parameters are required when declaring a DOUBLE PRECISION data type.

Syntax	double precision
Corresponding Compile-Time Java Type	java.lang.Double
JDBC Metadata Type (java.sql.Types)	DOUBLE
Default Value	Null
Minimum Value	4.9E-324
Maximum Value	1.7976931348623157E308
Size	8
Maximum Size	N.A

Examples of DOUBLE PRECISION Valid Values

4567890123456789012345

-1267890123456789012

Boolean

The **BOOLEAN** data type accepts a single value that can be TRUE or FALSE. No parameters are required when declaring a BOOLEAN data type.

Syntax	boolean
Corresponding Compile-Time Java Type	java.lang.Boolean
JDBC Metadata Type (java.sql.Types)	BOOLEAN
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	1
Maximum Size	N.A

Date-time Type

The date-time data type can be DATE, TIME or TIMESTAMP.

Syntax

<datetime type> ::=

DATE

| TIME [<left paren> <time precision> <right paren>]

| TIMESTAMP [<left paren> <timestamp precision> <right paren>]

Date

The **DATE** data type accepts date values, consisting of Year, Month, and Day. Date values should be specified in the form YYYY-MM-DD.

Month values must be between 1 and 12.

Day values should be between 1 and 31 depending on the month and *Year values* should be between 0 and 9999.

Values assigned to the DATE data type should be enclosed in single quotes, preceded by the keyword DATE.

Syntax	date
Corresponding Compile-Time Java Type	java.sql.Date
JDBC Metadata Type (java.sql.Types)	DATE
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	10
Maximum Size	N.A

Example

DATE '1999-04-04'.

Time

The **TIME** data type accepts time values, consisting of Hours, Minutes, and Seconds. Time values should be specified in the form HH:MM:SS.

The minutes and seconds values must be two digits. *Hour values* should be between zero 0 and 23. *Minute values* should be between 00 and 59 and *Second values* should be between 00 and 59. Values assigned to the TIME data type should be enclosed in single quotes, preceded by the keyword TIME.

Syntax	time[(n)]
Corresponding Compile-Time Java Type	java.sql.Time
JDBC Metadata Type (java.sql.Types)	TIME
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	N.A
Maximum Size	9

Example

TIME '07:30:00'.

Time Stamp

The **TIMESTAMP** data type accepts timestamp values, which are a combination of a DATE value and a TIME value. Timestamp values should be specified in the form YYYY-MM-DD HH:MM:SS.

There is a space separator between the date and time portions of the timestamp. *Month values* must be between 1 and 12. *Day values* should be between 1 and 31 depending on the month and *Year values* should be between 0 and 9999. *Hour values* should be between zero 0 and 23. *Minute values* should be between 00 and 59 and *Second values* should be between 00 and 59. Values assigned to the TIMESTAMP data type should be enclosed in single quotes, preceded by keyword TIMESTAMP.

Syntax	timestamp[(n)]
Corresponding Compile-Time Java Type	java.sql.TimeStamp
JDBC Metadata Type (java.sql.Types)	TIMESTAMP
Default Value	Null
Minimum Value	N.A
Maximum Value	N.A
Size	N.A
Maximum Size	9

Example

TIMESTAMP '1999-04-04 07:30:00'.

If the values are not in the specified ranges then Daffodil DB will convert them into valid values. Like if date is specified as 2001-12-31 then it will become 2002-01-01.

Domain Name

It is the name of the domain which represents data type with all its properties and constraints for value. Domains are defined, mainly when same properties of the data types are required frequently.

Literals

Literals are a type of expression that specifies a constant value (they are also called constants). There are various types of literals like Numeric Literal, Character String Literal, and Date-Time Literal etc.

<Literals>::=

<Character String Literal>

| <Date-Time Literal>

| <Numeric Literal>

| <Boolean Literal>

Character String Literal

The **Character String Literal** is a constant text literal. It can contain any alphabetic character, digit and special characters. The Character String Literal should be enclosed within single quotes. With the help of the quotes, database engine understands and treats it as constant.

Syntax

<character string literal>::= <quote> <text literal identifier body> <quote>

Examples of valid Character String Literal

'abd323'

'122djcvdj^***^*'

(')00000('

Examples of invalid Character String Literal

'cbcbmm

“scbdcdcb

Numeric Literal

Numeric Literal is used to specify a valid integer, double etc in expressions, SQL functions, and SQL statements.

Syntax

<Numeric Literal>::=

<Rep Digit>

<period> <Rep Digit>

| <Rep Digit> <period> <Rep Digit>

| <Rep Digit> {e | E} <Rep Digit>

<Rep digit>::= <digit>

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

A digit is one of 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9.

e or E indicates that the number is specified in scientific notation. The digits after E specify exponent and digits before E specify mantissa.

Examples of <Rep Digit> valid values

0

22

33

Examples of <Period> <Rep Digit> valid values

23

3

Example of <Rep Digit> <period> <Rep Digit> valid values

22.22

2.33

33.3

Example of <Rep Digit> E <Rep Digit> valid values

2E3

2E34

Date-Time Literal

Date-Time Literal is used to specify a date literal, time literal and timestamp literal in expressions, SQL Statements and Functions.

Syntax

<Date Time Literal>::=

<Date Literal>

<Time Literal>

<Time Stamp Literal>

Date Literal

Date Literal is used to specify a constant Date value in a specific format. This date value can be used in SQL Statements and Functions.

Format for specifying a date literal is: DATE 'YYYY-MM-DD'

Syntax

DATE '<unsigned integer> <hyphen> <unsigned integer> <hyphen> <unsigned integer>'

<unsigned integer>::= <Rep Digit>

<Rep Digit>::= <digit> . . <hyphen>::= -

Examples of valid Date Literals

DATE '2002-06-28'

DATE '2001-06-28'

Time Literal

Time Literal is used to specify a constant Time value in a specific format. This Time value can be used in SQL Statements and Functions.

Format for specifying a time literal is: TIME 'HH:MM:SS'

Syntax

TIME '<unsigned integer> <colon> <unsigned integer> <colon> <unsigned integer>'

<unsigned integer>::= <Rep Digit>

<Rep Digit>::= <digit> . . .

<colon>::= :

Examples of valid Time Literals

TIME '12:06:12'

TIME '13:06:28'

TimeStamp Literal

TimeStamp Literal is used to specify a constant TimeStamp value in a specific format. This TimeStamp value can be used in SQL Statements and Functions.

Format for specifying a TimeStamp literal is: TIMESTAMP 'YYYY-MM-DD HH:MM:SS'

Syntax

TIMESTAMP '<unsigned integer> <hyphen> <unsigned integer> <hyphen> <unsigned integer>
<space> <unsigned integer> <colon> <unsigned integer> <colon> <unsigned integer>'

<unsigned integer>::= <Rep Digit>

<Rep Digit>::= <digit>

<colon>::= :

<space>::=

Example of valid Timestamp Literals

TIMESTAMP '2002-06-28 12:06:12'

TIMESTAMP '2001-06-28 12:06:28'

Boolean Literal

Boolean Literal is a single constant value specifying TRUE or FALSE.

[`<Boolean Literal>::=`](#)

[`TRUE`](#)

[`FALSE`](#)

Examples of valid Boolean Literals

`TRUE`

`FALSE`

Functions

Daffodil DB provides few built-in functions to perform in-statement operations while querying or inserting data into a database. Functions are a type of SQL expression that return a value based on the argument they are supplied.

<Functions>::=<Numeric Functions>

<Date Time Functions>

<String Functions>

<System Functions>

<Aggregate Functions>

<Special Functions>

Numeric Functions

Numeric Functions accept numeric expression as input and return numeric values as output. There are various numeric functions like **ATAN**, **TAN**, and **SQRT** etc.

Date Time Functions

Date Functions accept Date Time Expression as input and return numeric or string values as output according to the return type of the function. There are various Date Time Functions like **DAYNAME**, **YEAR** and **MONTH** etc.

String Functions

String Functions accept String Expression as input and return string values as output. There are various String Functions like **SUBSTRING**, **LTRIM** and **RTRIM** etc.

System Functions

System Functions accept nothing as argument and return numeric or string values as output according to the return type of the function. There are various System Functions like **DATABASE** and **USER** etc.

Aggregate Functions

Aggregate Functions accept Expression as input and return numeric values as output. Aggregate Functions operate on group of records to perform aggregate operations. There are various Aggregate Functions like **SUM**, **AVG** and **MAX** etc.

Special Functions

There is only single Special Function, which takes numeric value as input. The function is **Top** Function. Functionality performed by Top Function is to select the topmost specified rows from Result Set as an output.

If you call a built-in function with an argument of a data type other than the data type expected by the built-in function, Daffodil DB implicitly converts the argument to the expected data type before performing the required operation.

Numeric Functions

Numeric Functions are special built-in functions for specific purposes. Numeric Functions either take zero, one or more numeric expressions as input. These functions act as special operators in the database identified by keywords in the database. Various functions are performed by these special operators like square root, floor, ceiling etc.

Syntax

<Numeric Functions> ::=

<absolute value expression>

<modulus expression>

<sine function>

<power function>

<round function>

<sqrt function>

<truncate function>

<floor function>

<ceiling function>

<log function>

<exp function>

<cos function>

<tan function>

<cot function>

<acos function>

<asin function>

<atan function>

<degrees function>

<radians function>

<pi function>

<atan2 function>

<rand function>

<sign function>

Absolute Value Expression

The absolute value expression is used to take absolute value of a numeric expression passed as an argument to it.

Syntax

ABS <left paren> <numeric expression> <right paren>

Numeric Expression is passed as input to the absolute function. The function calculates and returns an absolute value of any given expression.

If the argument is not negative, the argument is returned.

If the argument is negative, the negation of the argument is returned.

Example

Select ABS (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing absolute value of column 'ParentMarksCarryingForward' under column heading 'C'

Result

C
0
25
20
20

Modulus Value Expression

The modulus value expression is used to take modulus value of a numeric expression, dividend with divisor being numeric expression are passed as arguments.

Syntax

```
MOD <left paren> <numeric expression dividend> <comma>  
<numeric expression divisor> <right paren>  
<numeric expression dividend> ::= <numeric expression>  
<numeric expression divisor> ::= <numeric expression>
```

Numeric Expression dividend passed as input to modulus function is divided by Numeric Expression divisor and remainder is calculated and returns as the modulus value of an expression.

Example

Select MOD (ParentMarksCarryingForward, 2) as C from Exam

The above query results in a Result Set containing modulated value of column 'ParentMarksCarryingForward' under column heading 'C'.

Result

C
0.0
1.0
0.0
0.0

Sine Function

The sine function is used to calculate sine value of a numeric expression passed as an argument.

Syntax

`SIN <left paren> <numeric expression> <right paren>`

Numeric Expression is passed as input to sine function. The function calculates and returns the trigonometric sine of an angle. Argument passed is: Angle in Radians.

Example

Select SIN (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing trigonometric sine value of column 'ParentMarksCarryingForward' under column heading 'C'

Result

C
0.0
-0.13235175009777303
0.9129452507276277
0.9129452507276277

Power Function

The power function is used to return value of the first argument raised to the power of the second argument.

Syntax

`POWER <left paren> <numeric expression> <comma> <numeric expression> <right paren>`

If the second argument is positive or negative zero, then the result is 1.0.

If the second argument is 1.0, then the result is the same as the first argument.

Example

Select Power (ParentMarksCarryingForward, 2) as C from Exam

The above query results in a Result Set containing value of the column 'ParentMarksCarryingForward' raised to the power 2 under column heading 'C'.

Result

C
0.0
625.0
400.0
400.0

Rand Function

Rand Function generates a random number using numeric expression passed as the initial seed.

Syntax

RAND <left paren> <numeric expression> <right paren>

Numeric Expression is passed as input to a random function. The function calculates and returns a new random number generated using the argument passed.

Example

Select RAND (ParentMarksCarryingForward) as C from Exam where ParentMarksCarryingForward > 0

The above query returns a random number using 'ParentMarksCarryingForward' as initial seed under column heading 'C'.

Result

C
0.7315948009490967
0.7320427298545837
0.7320427298545837

SQRT Function

SQRT Function calculates square root of a numeric expression passed as an argument.

Syntax

SQRT <left paren> <numeric expression> <right paren>

Numeric Expression is passed as an input to the sqrt function. The function calculates and returns square root of the numeric expression.

Example

Select SQRT (ParentMarksCarryingForward) as C from Exam

The above query returns square root of 'ParentMarksCarryingForward' under column heading 'C' in the Result Set.

Result

C
0.0
5.0
4.47213595499958
4.47213595499958

TRUNCATE Function

TRUNCATE Function truncates the number (first argument numeric expression) to (second argument numeric expression) places.

Syntax

TRUNCATE <left paren> <numeric expression> <comma> <numeric expression> <right paren>

First Argument Numeric Expression is the number to be truncated.

Second Argument Numeric Expression is the places to which it is to be truncated.

Example

Select TRUNCATE (ParentMarksCarryingForward, 1) as C from Exam where ParentMarksCarryingForward > 0

The above query returns value of column 'ParentMarksCarryingForward' truncated to 1 place under column heading 'C' in the Result Set.

Result

C
25
20
20

FLOOR Function

The Floor function returns the largest (closest to positive infinity) double the value that is not greater than the argument and is equal to a mathematical integer.

Syntax

FLOOR <left paren> <numeric expression> <right paren>

If the argument value is already equal to a mathematical integer, then the result is the same as the argument, it means Largest integer <= number.

Argument passed is a Numeric Expression.

Example

Select FLOOR (ParentMarksCarryingForward) as C from Exam where ParentMarksCarryForward > 0

Result

C
25
20
20

CEILING Function

The Ceiling function returns the smallest (closest to negative infinity) double the value that is not lesser than the argument and is equal to a mathematical integer.

Syntax

CEILING <left paren> <numeric expression> <right paren>

If the argument value is already equal to a mathematical integer, then the result is the same as the argument, it means Smallest integer \geq number.

Argument passed is a Numeric Expression.

Example

Select CEILING (ParentMarksCarryingForward) as C from Exam where ParentMarksCarryingForward > 0

Result

C
25
20
20

LOG Function

The Log function returns the natural logarithm (base e) of double the value passed as an argument.

Syntax

LOG <left paren> <numeric expression> <right paren>

If the argument is less than zero, then the result is **NaN**.

If the argument is positive infinity, then the result is positive **infinity**.

If the argument is positive zero or negative zero, then the result is negative **infinity**.

Argument passed is a Numeric Expression.

Example

Select LOG (-23) as C from Post

Result

C
NaN
NaN
NaN

Select LOG (23) as C from Post

Result

C
3.1354942159291497
3.1354942159291497
3.1354942159291497

EXP Function

The EXP function returns the exponential number e (i.e., 2.718...) raised to the power of double the value passed as an argument.

Syntax

EXP <left paren> <numeric expression> <right paren>

Exponential function of an argument passed.

Argument passed is a Numeric Expression.

Example

Select EXP (ParentMarksCarryingForward) as C from Exam where ParentMarksCarryingForward > 0

Result

C
7.200489933738588E10
4.8516519540979037E8
4.8516519540979037E8

COS Function

The COS function returns trigonometric cosine of an angle.

Syntax

COS <left paren> <numeric expression> <right paren>

If the argument is **NaN** or infinity, then the result is **NaN**.

Argument passed is Angle in Radians.

Example

Select COS (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing trigonometric COS value of the column 'ParentMarksCarryingForward' under the column heading 'C'

Result

C
1.0
-0.9912028118634736
0.40808206181339196
0.40808206181339196

TAN Function

The TAN function returns the trigonometric tangent of an angle.

Syntax

TAN <left paren> <numeric expression> <right paren>

If the argument is **NaN** or infinity, then the result is **NaN**.

Argument passed is Angle in Radians.

Example

Select TAN (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing trigonometric tan value of the column 'ParentMarksCarryingForward' under column heading 'C'.

Result

C
0.0
-0.13352640702153587
2.237160944224742
2.237160944224742

COT Function

The COT function returns the trigonometric Cotangent of an angle in radians.

Syntax

COT <left paren> <numeric expression> <right paren>

If the argument is **NaN** or infinity, then the result is **NaN**.

Argument passed is Angle in Radians.

Example

Select COT (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing trigonometric COT value of the column 'ParentMarksCarryingForward' under column heading 'C'.

Result

C
Infinity
-7.489155308722675
0.44699510899489167
0.44699510899489167

ACOS Function

The ACOS function returns the arc cosine of an angle, in the range of 0.0 through pi.

Syntax

ACOS <left paren> <numeric expression> <right paren>

If the argument is **NaN** or its absolute value is greater than 1, then the result is **NaN**.

Parameter passed is double the value whose arc cosine is to be returned.

Function returns arc cosine of the argument.

Example

Select ACOS (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing trigonometric arc COS value of column 'ParentMarksCarryingForward' under column heading 'C'.

Result

C
1.5707963267948966
NaN
NaN
NaN

ASIN Function

The ASIN function returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$.

Syntax

ASIN <left paren> <numeric expression> <right paren>

If the argument is **NaN** or its absolute value is greater than 1, then the result is **NaN**.

Parameter passed is a double value whose arc sine is to be returned.

Function returns arc sine of the argument.

Example

Select ASIN (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing trigonometric arc sin value of the column 'ParentMarksCarryingForward' under column heading 'C'

Result

C
0.0
NaN
NaN
NaN

ATAN Function

The ATAN function returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$, where value of π is 3.14

Syntax

ATAN <left paren> <numeric expression> <right paren>

If the argument is **NaN** or infinity, then the result is **NaN**.

Argument passed is double the value whose arc tangent is to be returned.

Returns arc tangent of the argument.

Example

Select ATAN (ParentMarksCarryingForward) as C from Exam

The above query results in a Result Set containing trigonometric arc tan value of the column 'ParentMarksCarryingForward' under column heading 'C'.

Result

C
0.0
1.5308176396716067
1.5208379310729538
1.5208379310729538

DEGREES Function

The Degrees function converts an angle measured in radians to the equivalent angle measured in degrees.

Syntax

DEGREES <left paren> <numeric expression> <right paren>

Argument passed is an angle, in radians.

Function returns the measurement of the angle passed in degrees.

Example

Select DEGREES (10) as C from Post

Result

C
572.9577951308232
572.9577951308232
572.9577951308232

RADIANS Function

The Radians function converts an angle measured in degrees to the equivalent angle measured in radians.

Syntax

RADIANS <left paren> <numeric expression> <right paren>

Argument passed is an angle, in degrees.

Function returns measurement of the angle passed in radians.

Example

Select RADIANS (180) as RADIANS from Exam

Result

RADIANS
3.14159265358979323
3.14159265358979323
3.14159265358979323
3.14159265358979323

PI Function

The PI function Double the value that is closer than any other number to pi, the ratio of the circumference of a circle to its diameter.

Value of PI is 3.141592653589793.

Syntax

PI <left paren> <right paren>

Function returns the value of PI.

Example

Select PI () as PI from Exam

Result

PI
3.141592653589793
3.141592653589793
3.141592653589793
3.141592653589793

ATAN2 Function

The ATAN2 function converts the rectangular coordinates (b, a) to polar (r, theta).

This method computes the phase theta by computing the arc tangent of a/b in the range of -pi to pi.

Syntax

ATAN2 <left paren> <numeric expression> <comma> <numeric expression> <right paren>

First Argument passed to a function is double the value, i.e. b.

Second Argument passed to a function is double the value, i.e. a.

Function returns theta component of the point (r, theta) in polar coordinates that corresponds to the point (b, a) in Cartesian coordinates.

Example

Select ATAN2 (360, 45) as C from Exam

Result

C
1.446441332248135
1.446441332248135
1.446441332248135
1.446441332248135

ROUND Function

The Round function is used to round a number towards its "nearest possible neighbor". Round Function rounds a number to places.

Syntax

ROUND <left paren> <numeric expression> <comma> <numeric expression> <right paren>

First Argument is the number to be rounded. Second Argument is places to which the number should be rounded off.

Example

Select ROUND (5.671495, 4) as C from Post

Result

C
5.6715
5.6715
5.6715

SIGN Function

The Sign Function determines the sign of the numeric expression passed as an argument.

Syntax

SIGN <left paren> <numeric expression> <right paren>

Numeric Expression is passed as an argument.

If Argument passed is less than zero (number < 0) then the result is negative (-1)

If Argument passed is equal to zero (number == 0) then the result is zero (0).

If Argument passed is greater than zero (number > 0) then the result is positive (1).

Example

Select SIGN (-1) as C from Post

Result

C
-1
-1
-1

Date Time Functions

Date Time Functions are special built-in functions for specific purposes. Date Time Functions either take zero, one or more Date Time Expressions as Input. These functions act as special operators in the databases and are identified by keywords in the database. Various Functions are performed by these special operators like monthname, dayOfMonth and dayOfWeek etc.

The declared data type of expression used in Date Time Functions is DATE, TIME or TIMESTAMP.

Syntax

<Date Time Functions> ::=

<dayname function>

| <dayofmonth function>

| <dayofweek function>

| <dayofyear function>

| <week function>

| <month function>

| <year function>

| <monthname function>

| <hour function>

| <minute function>

| <second function>

| <timestampadd function>

| <timestampdiff function>

| <Curdate function>

| <Curtime function>

| <Curtimestamp function>

| <Date function>

| <Time function>

DAYNAME Function

The DAYNAME function returns a character string representing day component of the date, name for the day, which is specific to the data source.

For Example: Data Source is (Monday, Tuesday, Wednesday,..... Sunday)

Syntax

DAYNAME <left paren> <expression> <right paren>

Example

Select DAYNAME (DateOfJoining) as DateOfJoining from Teacher

Result

DateOfJoining
WEDNESDAY
TUESDAY
FRIDAY
SUNDAY
SATURDAY
TUESDAY
TUESDAY
SUNDAY

DAYOFMONTH Function

An integer from 1 to 31 representing day of the month in a date is returned upon calling the DAYOFMONTH function.

Syntax

DAYOFMONTH <left paren> <expression> <right paren>

Example

Select DayOfMonth (DateOfJoining) from Teacher

Result

DayOfMonth(DateOfJoining)
17
1
25
25
29
15
15
25

DAYOFWEEK Function

The DAYOFWEEK function returns an integer from 1 to 7 representing day of the week in a date; 1 indicates that Sunday is returned after execution of this function.

Syntax

DAYOFWEEK <left paren> <expression> <right paren>

Example

Select DayOfWeek (DateOfJoining) from Teacher

Result

DayOfWeek(DateOfJoining)
4
3
6
1
7
3
3
1

DAYOFYEAR Function

An integer from 1 to 366 representing day of the year in a date is returned on executing the DAYOFYEAR function.

Syntax

DAYOFYEAR <left paren> <expression> <right paren>

Example

Select DayOfYear (DateOfJoining) from Teacher

Result

DayOfYear(DateOfJoining)
108
182
268
298
241
258
258

WEEK Function

An integer from 1 to 53 representing week of the year in a date is returned on executing the WEEK function.

Syntax

WEEK <left paren> <expression> <right paren>

Example

Select Week (DateOfJoining) from Teacher

Result

Week(DateOfJoining)
16
27
39
44
35
38
44

MONTH Function

An integer from 1 to 12 representing month component of a date is returned on executing the MONTH function.

Syntax

MONTH <left paren> <expression> <right paren>

Example

Select MONTH (DateOfJoining) from Teacher

Result

MONTH(DateOfJoining)
4
7
9
10
8
9
9
10

YEAR Function

An integer representing year component of a date is returned on executing the YEAR function.

Syntax

YEAR <left paren> <expression> <right paren>

Example

Select YEAR (DateOfJoining) from Teacher

Result

YEAR(DateOfJoining)
1996
1997
1998
1998
1998
1998
1998
1998
1998

MONTHNAME Function

A character string representing month component of a date is returned on executing the MONTHNAME function.

The name for the month is specific to the data source.

For Example Data Source is: (January, February, March, December)

Syntax

MONTHNAME <left paren> <expression> <right paren>

Example

Select MONTHNAME (DateOfJoining) from Teacher

Result

MONTHNAME(DateOfJoining)
APRIL
JULY
SEPTEMBER
OCTOBER
SEPTEMBER
OCTOBER
OCTOBER

HOURL Function

An integer from 0 to 23 representing hour component of time is returned on executing the HOUR function.

Syntax

HOUR <left paren> <expression> <right paren>

Example

Select HOUR (CURTIME ()) from Teacher

Result

HOUR(CURRENT_TIME)
18
18
18
18

MINUTE Function

An integer from 0 to 59 representing minute component of time is returned on executing the MINUTE function.

Syntax

MINUTE <left paren> <expression> <right paren>

Example

Select MINUTE (CURTIME ()) from Teacher

Result

MINUTE(CURRENT_TIME)
32
32
32
32

SECOND Function

An integer from 0 to 59 representing second component of time is returned on executing the SECOND function.

Syntax

SECOND <left paren> <expression> <right paren>

Example

Select SECOND (CURTIME ()) from Teacher

Result

SECOND(CURRENT_TIME)
34
34
34
34

TIMESTAMPADD Function

The TIMESTAMPADD function returns the timestamp calculated by adding count number of the interval(s) to timestamp. An interval may be one of the following:

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, or SQL_TSI_YEAR

Syntax

<timestamp denometers> ::=

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

<timestampadd function> ::=

`TIMESTAMPADD <left paren> <timestamp denometers> <comma> <expression1><comma>
<expression2> <right paren>`

Timestamp Denometers

This argument specifies interval of the Time Stamp to which count is to be added. There are various types of intervals like:

SQL TSI FRAC SECOND

Interval is Seconds Fractional Part of Time Stamp.

SQL TSI SECOND

Interval is Seconds Part of Time Stamp.

SQL TSI MINUTE

Interval is Minutes Part of Time Stamp.

SQL TSI HOUR

Interval is Hours Part of Time Stamp.

SQL TSI DAY

Interval is Day Part of Time Stamp.

SQL TSI WEEK

Interval is Week Part of Time Stamp, in which Week timestamp lies.

SQL TSI MONTH

Interval is Month of Time Stamp.

SQL TSI QUARTER

Interval is Quarter Part of Time Stamp. The year in which quarter timestamp lies.

SQL TSI YEAR

Interval is Year Part of Time Stamp.

Expression1

This argument expression is a numeric value, which is to be added to a specified interval of Time Stamp. The declared data type of expression is numeric.

Expression2

Expression2 represents a Time Stamp to which required operations are to be performed. The declared data type of expression is TimeStamp.

Return type of the function is an object of modified TimeStamp.

Example

Select `TIMESTAMPADD (SQL_TSI_SECOND, 34, TIMESTAMP '2002-06-28 17:12:12')` AS "TIME
ADDITION IN SECONDS" from Exam

Result

TIME ADDITION IN SECONDS
2002-06-28 17:12:46.0
2002-06-28 17:12:46.0
2002-06-28 17:12:46.0
2002-06-28 17:12:46.0

Select **TIMESTAMPADD** (SQL_TSI_HOUR, 34, **TIMESTAMP** '2002-6-28 17:12:12') AS
"TIME ADDITION IN HOURS" from Exam

Result

TIME ADDITION IN HOURS
2002-06-30 03:12:12.0
2002-06-30 03:12:12.0
2002-06-30 03:12:12.0
2002-06-30 03:12:12.0

TIMESTAMPDIFF Function

The **TIMESTAMPDIFF** function returns an integer representing the number of interval by which timestamp2 is greater than timestamp1.

Interval may be one of the following:

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, or
SQL_TSI_YEAR

Syntax

TIMESTAMPDIFF <left paren> <timestamp denometers> <comma> <expression1> <comma>
<expression2> <right paren>

Timestamp Denometers

This argument specifies the interval of Time Stamp whose count is to be returned. For various types of intervals, refer to the previous function.

Expression1

Expression1 represents Time Stamp, which is subtracted from the second argument. Declared data type of an expression is TimeStamp.

Expression2

Expression2 represents Time Stamp from which first argument is subtracted. Declared data type of an expression is TimeStamp.

Return type of a function is count of the intervals.

Example

Select TIMESTAMPDIFF (SQL_TSI_SECOND, TIMESTAMP '2002-06-28 17:12:12',
TIMESTAMP '2002-05-28 7:7:7') AS "TIME DIFFERENCE IN SECONDS" from Exam

Result

TIME DIFFERENCE IN SECONDS
-2714705
-2714705
-2714705
-2714705

Select TIMESTAMPDIFF (SQL_TSI_HOUR, TIMESTAMP '2002-06-28 17:12:12', TIMESTAMP
'2002-06-28 7:7:7') AS "TIME DIFFERENCE IN HOURS" from Exam

Result

TIME DIFFERENCE IN HOURS
-10
-10
-10
-10

CURDATE Function

The CURDATE function returns the Current Date of an SQL Session. The declared data type of the function is DATE.

Syntax

CURDATE <left paren> <right paren>

Example

Select CURDATE () as CURDATE from Exam

Result

CURDATE
2003-10-30
2003-10-30
2003-10-30
2003-10-30

CURTIME Function

The CURTIME function returns Current Time of an SQL Session. The declared data type of the function is TIME.

Syntax

CURTIME <left paren> <right paren>

Example

Select CURTIME () as CURTIME from Exam

Result

CURTIME
18:32:23
18:32:23
18:32:23
18:32:23

CURTIMESTAMP Function

Returns Current Timestamp of an SQL Session. The declared data type of the function is timestamp.

Syntax

CURTIMESTAMP <left paren> <right paren>

Example

Select CURTIMESTAMP () as CURTIMESTAMP from Exam

Result

2002-06-28 17:12:46.0
2002-06-28 17:12:46.0
2002-06-28 17:12:46.0
2002-06-28 17:12:46.0

DATE Function

Date function extract date from timestamp. Argument passed to the function is timestamp and declared data type of the function is DATE.

Syntax

DATE <left paren> <expression> <right paren>

Example

Select DATE (Timestamp '2002-06-30 12:12:12') as DateOfJoining from Teacher

Result

DateOfJoining
2002-06-30
2002-06-30
2002-06-30
2002-06-30

TIME Function

Time functions extract time from timestamp. Argument passed to the function is timestamp and Declared Data type of the function is time

Syntax

TIME <left paren> <expression> <right paren>

Example

Select TIME (Timestamp '2002-06-30 12:12:12') as TimeOfJoining from Teacher

Result

TimeOfJoining
17:12:12
17:12:12
17:12:12
17:12:12

String Functions

String Functions are special built-in functions for specific purposes. String Functions either take zero, one or more String expressions as Input. These functions act as special operators in the databases, identified by keywords in a database. Various Functions are performed by these special operators, for e.g. lcase, ucase, right and left functions and so on.

Syntax

<String Functions> ::=

<ASCII value method>

| <left function>

| <right function>

| <space function>

| <replace function>

| <repeat function>

| <soundex function>

| <insert function>

| <difference function>

| <concat function>

| <locate function>

| <lcase function>

| <ucase function>

| <ltrim function>

| <rtrim function>

| <char function>

| <length function>

| <substring function>

| <EqualsCaseSensitive function>

ASCII Value Function

The String function ASCII returns an Integer representing ASCII code value of the leftmost character in a string.

Syntax

ASCII <left paren> <String Expression> <right paren>

Examples

Select ASCII (SubjectName) AS "ASCII VALUE" from Subject

The above query retrieves an ASCII code value of the leftmost character in the SubjectName from a Subject table.

Result

ASCII VALUE
66
69
77
83
83

Left Function

The Left Function returns count of the leftmost characters from a string.

Syntax

LEFT <left paren> <String Expression> <comma> <string length> <right paren>
<string length> ::= <unsigned integer>

First Argument String Expression is a String from which number of characters is to be retrieved.

Second Argument String Length is the number of the characters to be retrieved.

Examples

Select left (StudentName, 4) as "LEFT MOST" from Student

The above query retrieves 4 leftmost characters for each student name from the '*Student*' table.

Result

LEFT MOST
Cath
John
Cath
John
Woll
Vali
Lieb
Will
Tove
Wink

Right Function

The Right Function returns count of the rightmost characters from a string.

Syntax

RIGHT <left paren> <String Expression> <comma> <string length> <right paren>
<string length> ::= <unsigned integer>

First Argument String Expression is a String from which number of characters is to be retrieved.

Second Argument String Length is the number of the characters to be retrieved.

Examples

Select right (StudentName, 3) as "RIGHT MOST" from Student

The above query retrieves 3 rightmost characters for each student name from the *Student* table.

Result

RIGHT MOST
ine
ohn
the
ohn
oll
ine
big
ams
era
eld

Space Function

The String function Space returns a character string consisting of a number of specified spaces.

Syntax

SPACE <left paren> <string length> <right paren>

Examples

Select space (5) AS "SPACES" from Post

The above query retrieves the String containing 5 spaces from the Post table

Result

SPACES

Replace Function

The String function Replace, replaces all the occurrences of string2 in string1 with String3.

Syntax

REPLACE <left paren> <String Expression> <comma> <String Expression> <comma> <String Expression> <right paren>

Second String Expression is a substring in the First String Expression to be replaced with the third String Expression.

Examples

Select replace (TeacherName,'Mr.','Shri') AS REPLACEMENT from Teacher

The above query replaces all the occurrences of String 'Mr.' with String 'Shri' in the TeacherName Column of the Teacher Table.

Result

REPLACEMENT
Shri Agregado
Shri Brumfield
Ms. McKelvey
Shri Everett
Shri Verstrepen
Shri Haight
Ms. Hartenfeld
Shri Henry

Repeat Function

The String function Repeat returns a character string formed by repeating the number of string count times.

Syntax

REPEAT <left paren> <String Expression> <comma> <numeric expression> <right paren>

Examples

Select repeat (PostName, 2) AS REPEAT_TWO_TIMES from Post

The above query retrieves post names by repeating the string count two times.

Result

REPEAT_TWO_TIMES
PrincipalPrincipal
Vice PrincipalVice Principal
TeacherTeacher

Soundex Function

The scalar function SOUNDEX returns a character string, which is data source-dependent, representing sound of words in a string; it could be a four-digit SOUNDEX code, a phonetic representation of each word, etc.

Syntax

SOUNDEX <left paren> <String Expression> <right paren>

Examples

Select soundex (StudentName) AS SOUNDEX_CODE from Student

The above query retrieves a four-digit code for each student name from the *Student* table.

Result

SOUNDEX_CODE
C365
J500
C300
J500
W400
V450
L120
W452
T160
W521

Insert Function

The string function INSERT returns a character string formed by deleting length of characters from string1 beginning at start, and inserting string2 into string1 in the beginning.

Syntax

INSERT <left paren> <String Expression> <comma> <numeric expression>
<comma> <numeric expression> <comma> <String Expression> <right paren>

First String Expression Argument is a String in which another String needs to be inserted.

Start is the index from which Length characters need to be deleted.

Second String Expression is a String to be inserted at the index Start.

Examples

Select insert (TeacherName, 1, 3, 'Shri') AS REPLACEMENT From Teacher

The above query replaces 3 characters starting at index 1 and replaces with it String 'Shri'.

Result

REPLACEMENT
Shri Agregado
Shri Brumfield
Shri McKelvey
Shri Everett
Shri Verstrepn
Shri Haight
Shri Hartenfeld
Shri Henry

Difference Function

The string function **DIFFERENCE** returns an Integer indicating the difference between values returned by the function **SOUNDEX** for *string1* and *string2*.

Syntax

DIFFERENCE <left paren> <String Expression> <comma> <String Expression> <right paren>

Examples

Select difference (StudentName, 'John') AS "DIFFERENCE" from Student

The above query retrieves difference between the values returned by the function **SOUNDEX** for each student name and student address from the *Student* table.

Result

DIFFERENCE
0
4
2
4
2
1
1
0
1
1

Concat Function

The string function CONCAT returns a Character string formed by appending string2 to string1; if a string is null, then the result is DBMS dependent.

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.

Syntax

CONCAT <left paren> <String Expression> <comma> <String Expression> <right paren>

First String Expression is a String in which another String needs to be concatenated.

Second String Expression is a String to be concatenated with the First String Expression.

Examples

Select concat (StudentName, 'Calera') AS "NEW NAMES" from Student

The above query retrieves student names after appending 'Calera' with each student name from the *Student* table.

Result

NEW NAMES
Catherine Calera
John Calera
Cathe Calera
John Calera
Woll Calera
Valine Calera
Liebig Calera
Williams Calera
Tovera Calera
WinkField Calera

Locate Function

The string function LOCATE returns the location of the first occurrence of string1 in string 2, searching from the beginning of string 2.

If start is specified, then search begins from the position start. 0 is returned if string2 does not contain string1. If either string is null, LOCATE returns a null value.

Position 1 is the first character in string2.

Syntax

LOCATE <left paren> <String Expression> <comma> <String Expression> [<comma> <numeric expression>] <right paren>

First String Expression is the String in which another String is to be located.

Second String Expression is the String to be located in the First String Expression.

Examples

Select locate (StudentName,'John') AS "FIND JOHN" from Student

The above query retrieves location of 'John' from the *Student* table.

Result

FIND JOHN
0
1
0
1
0
0
0
0
0
0
0

Lcase Function

The string function LCASE returns the result after converting all the characters in a string to lowercase.

Syntax

LCASE <left paren> <String Expression> <right paren>

Examples

Select lcase (TeacherName) as "LOWER CASE NAMES OF TEACHERS" from Teacher

The above query retrieves the name of teachers in lowercase.

Result

LOWER CASE NAMES OF TEACHERS
mr. agregado
mr. brumfield
ms. mckelvey
mr. everett
mr. verstrepen
mr. haight
ms. hartenfeld
mr. henry

Ucase Function

The string function UCASE returns the result after converting all the characters in a string to uppercase

Syntax

UCASE <left paren> <String Expression> <right paren>

Examples

Select ucase (TeacherName) as "UPPER CASE NAMES OF TEACHERS" from Teacher

The above query retrieves the name of teachers in uppercase.

Result

UPPER CASE NAMES OF TEACHERS
MR. AGREGADO
MR. BRUMFIELD
MS. MCKELVEY
MR. EVERETT
MR. VERSTREPEN
MR. HAIGHT
MS. HARTENFELD
MR. HENRY

Ltrim Function

The String function LTRIM removes all the characters of a string with leading blank spaces.

Syntax

LTRIM <left paren> <String Expression> <right paren>

Examples

Select ltrim (StudentName) AS "NAMES AFTER LEFT TRIMMING" from Student

The above query retrieves name of students after removing the leading blank spaces from *Student* table.

Result

NAMES AFTER LEFT TRIMMING
Catherine
John
Cathe
John
Woll
Valine
Liebig
Williams
Tovera
WinkField

Rtrim Function

The String function RTRIM retrieves characters of a string with no trailing blanks.

Syntax

RTRIM <left paren> <String Expression> <right paren>

Examples

Select Rtrim (StudentName) AS "NAMES AFTER RIGHT TRIMMING" from Student

The above query retrieves names of students after removing trailing blank spaces from the Student table.

Result

NAMES AFTER RIGHT TRIMMING
Catherine
John
Cathe
John
Woll
Valine
Liebig
Williams
Tovera
WinkField

Char Function

The scalar function CHAR returns a Character with ASCII value code, where code is between 0 and 255

Syntax

CHAR <left paren> <numeric expression> <right paren>

Examples

Select char (65) AS "CHAR VALUE" from Post

Result

CHAR VALUE
A
A
A

Length Function

The string function LENGTH returns the number of characters in a string, excluding trailing blanks.

Syntax

LENGTH <left paren> <String Expression> <right paren>

String Expression is a String for which length is to be calculated.

Examples

Select length (TeacherName) AS LENGTH from Teacher

The above query retrieves a number of characters in the teacher name column from *Teacher* table.

Result

LENGTH
12
13
12
11
14

Substring Function

The String function SUBSTRING retrieves a character string formed by extracting length of characters from a string; beginning from start.

Syntax

SUBSTRING <left paren> <string expressions> <comma> <numeric expression>
<comma><numeric expression> <right paren>

<string expressions> ::= Any String

First String Expression Argument is a String from which a substring is to be retrieved.

First Numeric Expression Argument is the start index from where a substring is to be retrieved.

Second Numeric Expression Argument retrieves the number of characters to be retrieved.

Examples

Select substring (TeacherName, 4, 2) AS SUB_STRING from Teacher

The above query retrieves a substring from the teacher name starting at an index of 4 of length = 6.

Result

SUB_STRING
A
B
M
E
V
H
H
H

EqualsCaseSensitive Function

EqualsCaseSensitive retrieve true or false after checking the equality of two strings. It is a case sensitive function. It retrieve false if alphabets of one string are in different case from other.

Syntax

EQUALSCASESENSITIVE <left paren> <string expressions> <comma> <string expressions>
<right paren>

String expressions are parameters of equalscasesensitive function.

Examples

Select EqualsCaseSensitive (TeacherName, 'Mr. Agregado') AS STRING_EQUALITY from Teacher

The above example retrieves a substring from the teacher name starting at an index of 4 of length = 6.

Result

STRING_EQUALITY
True
false
false
false
false
false
false
false

System Functions

System Functions are certain built-in functions, which are used to perform activities like they are used to return current database name, current user name etc.

<System Functions>::=

<Current Database function>

| <user function>

| <ifnull function>

Current Database Function or CURRENT DATABASE

Current Database Function returns the name of current database in the SQL Session.

Syntax

DATABASE <left paren> <right paren>

Example

Select Database () as "DATABASE NAME" from Exam

Result

DATABASE NAME
school
school
school
school

User Function or CURRENT USER

The User function returns the name of a current user in the SQL Session.

Syntax

USER <left paren> <right paren>

Example

Select USER () from Exam

Result

User()
daffodil
daffodil
daffodil
daffodil

IFNULL Function

The IFNULL function performs a special task of checking the 'if' condition.

It checks for the null ness of the first expression. If value of the first expression is null then second expression is returned, otherwise value of first expression is returned.

Syntax

IFNULL <left paren> <expression1> <comma> <expression2> <right paren>

Example

Select IFNULL (postname,'not found') as POST_NAMES from post

Result

POST_NAMES
Principal
VicePrincipal
Teacher

Special Functions

There is only one special function namely **TOP** function. This function performs special purpose task. It is used to specify number of rows to be retrieved in a Result Set.

TOP Function

The TOP function enables us to control the number of rows to appear in the Result Set of a query result. We can specify the count 'n' to instruct query engine to retrieve only the specified topmost count of rows.

Syntax

TOP <left paren> <unsigned integer> <right paren>

Unsigned Integer

Unsigned Integer is any valid SQL integer, i.e. it should consist of digits (0-9).

This integer specifies the count of topmost rows to be retrieved in the Result Set. This integer can not be negative.

Example

Select TOP (5) floor ((marks*100)/500) as Percentage, StudentID, SubjectID from Marksrecord

Result

Percentage	StudentID	SubjectID
19	1	1
17	1	2
19	1	3
15	1	4
10	1	5

The query stated above is another form of **SELECT** queries, where we have used **TOP** function to list the top 5 students from the list. To display implementation of the mathematical expression in

SELECT statement, we have calculated the percentage taking maximum marks as 500. Also the column aliasing has been used in the query, where first column has been renamed to Percentage.

Aggregate Functions

Aggregate functions return a single result row based on groups of the rows, rather than on single rows. Aggregate functions can appear in select lists and in **ORDER BY** and **HAVING** clauses. They are commonly used with the **GROUP BY** clause in a **SELECT** statement, where Daffodil DB divides the rows of a queried table or view into groups. In a query containing **GROUP BY** clause, elements of the select list can be aggregate functions, **GROUP BY** expressions, constants, or expressions involving one of these.

Aggregate functions are applied to each group of rows and return a single result row for each group. In the absence of **GROUP BY** clause, Aggregate functions are applied in the select list to all the rows in the queried table or view. We use aggregate functions in the **HAVING** clause to eliminate groups from the output based on the results of the aggregate functions, rather than on the values of the individual rows of the queried table or view.

Aggregate functions are set of functions that apply over a set of column values and return a scalar value. Daffodil DB provides a number of Aggregate functions with each function operating over a set of column values and resulting in a single value.

Syntax

<set function specification> ::=

COUNT <left paren> <asterisk> <right paren>

| <general set function>

<general set function> ::=

<set function type> <left paren> <Aggregate Expression> <right paren>

<set function type> ::=

AVG | MAX | MIN | SUM

| COUNT

<Aggregate Expression> ::=

<Numeric Expression>

<Constant>

| <Column Reference>

Count

COUNT is an Aggregate function used to count number of records corresponding to a column or record. It takes a Numeric Expression or 'asterisk' (*) as its argument. Example

Select COUNT (*) from post.

Result

COUNT(*)

3

Result of the query mentioned above is simply count of the record in the Table *Post*.

Avg

AVG is an Aggregate function used to calculate average of the values corresponding to a column specified as its argument.

```
SELECT AVG (ABS (marks)) AS AVERAGE FROM MarksRecord GROUP BY ExamID
```

Result

AVERAGE

79.91666..

Sum

SUM is an Aggregate function used to calculate sum of all the values corresponding to a column specified as its argument.

```
Select SUM (marks*2) from MarksRecord where StudentID =5
```

Result

SUM(marks*2)

956

In the above example, result will be sum of all the marks multiplied by 2 corresponding to the student whose StudentId is 5.

Max/Min

MAX is an Aggregate function that finds the maximum value among all the available values corresponding to a column specified as its argument. On the contrary, MIN is an aggregate function that selects minimum value among all the available values corresponding to a column specified as its argument.

All aggregate functions except COUNT (*) and GROUPING ignore nulls. COUNT never returns Null, returns either a number or zero. For all the remaining aggregate functions, if data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns NULL. Nesting of Aggregate functions is also possible. For example, the following example calculates the average of the maximum marks of all the Exams held.

```
Select MAX (marks) from MarksRecord where ExamID <> 5
```

Result

MAX(marks)

99

In the above example, result will be Maximum marks in all the exams except that for which ExamID is 5.

Expression

An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value. An expression generally assumes the data type of its components. An Expression is a valid SQL expression (according to SQL-99 Standards) that can comprise of a Numeric Expression, Boolean Expression, String Expression or Expression Primary.

Syntax

<Expression> ::=

<Numeric Expression>

<Boolean Expression>

| <String Expression>

| <Expression Primary>

Numeric Expression

Numeric Expression is a type of expression that can be used to perform numeric operations.

Boolean Expression

Boolean Expression is a type of expression that returns a Boolean value. Boolean Precedence is to be applied properly while solving any Boolean Expression.

String Expression

String Expression represents a set of characters. Two String Expressions can be joined together through concatenation operator.

Expression Primary

Expression Primary can either be a SubQuery, Constant, Column Reference, Multi-Valued Expression or a Parenthesized Expression.

Expression Precedence

Precedence of operations from highest to lowest is:

(), ?,

unary + and -

*, /, || (concatenation)

binary + and -

NOT

AND

OR

?: (Conditional)

You can explicitly specify precedence by placing expressions within parentheses. An expression within parentheses is evaluated before any operations outside the parentheses are applied to it.

Example

(3+4)*9

In the above example although precedence of * is greater than +, but since parenthesis has highest precedence, first of all expression inside the parenthesis will be evaluated and then the result will be evaluated with *.

On the other hand in

3+4*9

First * will be evaluated and thereafter the result will be evaluated using + operator.

Numeric Expression

Numeric Expression is any kind of valid expression that contains plus sign, minus sign, asterisk and/or solidus in between two Numeric Expressions or it could be a simple factor. Numeric Expression is used to perform arithmetic operations. A valid plus or a minus sign can also be applied in a valid Numeric Expression. All the Numeric Expression will be evaluated by applying the proper Expression Precedence as mentioned above.

Syntax

<Numeric Expression> ::=

<term>

| <Numeric Expression> <minus sign> <term>

| <Numeric Expression> <plus sign> <term>

<term> ::=

<factor>

| <term> <asterisk> <factor>

| <term> <solidus> <factor>

<factor> ::=

[<sign>] <Numeric primary>

<Numeric Primary> ::=

<Expression Primary>

<Numeric Functions>

| <Date Time Functions>

<sign> ::= <plus sign> | <minus sign>

<plus sign> ::= +

<minus sign> ::= -

<asterisk> ::= *

<solidus> ::= /

Numeric Primary

A Numeric Primary can be a valid Expression Primary, valid Numeric Function or a valid Date Time Functions.

Term

A term in turn can be a valid factor or valid term asterisk / solidus factor.

Factor

A Factor can be a valid Expression Primary with or without sign.

Numeric Functions

A Numeric function is a set of Numeric functions like **SIN**, **COS**, **TAN**, **TRIM** etc.

Date Time Functions

These include several Date Time Functions like **DAYNAME**, **DAYOFWEEK** etc.

Sign

A sign could be either plus sign or minus sign.

Examples

$(a * b + c / d) + 2$ is a valid Numeric expression.

Here the expression inside the parenthesis will be evaluated first. Further the Expression inside the parenthesis will be treated itself as an Expression and Expression precedence will be properly applied over it. Further the result will be solved using the + operator.

In case of Numeric Expression, Multi-valued Expressions are not supported.

Boolean Expression

Boolean Expressions are any kind of valid expression that contains AND / OR in between two Boolean Expression, Boolean Primary or simply a Truth value. Boolean Expressions are used to perform Boolean operations that results in TRUE or FALSE value. Boolean expressions are allowed in a number of places, most notably in WHERE clauses, but also in check constraints and VALUES expressions. A Boolean expression can include a Boolean operator or operators. These are listed.

Syntax

<Boolean Expression> ::=

<boolean term>

| <boolean expression> OR <boolean term>

<boolean term> ::=

<boolean factor>

|<boolean term> AND <boolean factor>

<boolean factor> ::=

[NOT] <boolean test>

<boolean test> ::=

<boolean primary> [IS [NOT] <truth value>]

<truth value> ::=

TRUE | FALSE

<boolean primary> ::=

<parenthesized Boolean expression>

<Expression primary>

| <predicate>

<parenthesized Boolean expression> ::=

<left paren> <Boolean expression> <right paren>

Boolean Expression

A Boolean Expression can be a valid Boolean Primary, truth value or a valid Boolean Expression **AND/OR** Boolean term.

Boolean Term

A Boolean term in turn can be a valid Boolean factor or Boolean term **AND** Boolean factor.

Boolean factor

Boolean factor further could be any Boolean Test with or without NOT before it.

Boolean Primary

Boolean Primary further can be represented using parenthesized Boolean Expression, Expression Primary or a Predicate. Truth Value may be TRUE or FALSE.

Example 1

Select * from post where postID < 10 AND postID > 3 OR postName <> 'Principal'

Here in the above example *postID < 10 AND postID > 3 OR postName <> 'Principal'* is an example of valid Boolean Expression.

Result

PostID	PostName	PostRank
2	Vice Principal	2
3	Teacher	3

Example 2

Select * from post where TRUE

Here in the above example simply '**TRUE**' is also an example of valid Boolean Expression.

Result

PostID	PostName	PostRank
1	Principal	1
2	Vice Principal	2
3	Teacher	3

String Expression

String Expression is a valid Character Value Expression that further defines characters factor and/or character value expression concatenation operator character factor. Character Primary further could be any Expression Primary or a string value function. Concatenation operator is represented by '||' symbol and is used to join two character value Expression.

Syntax

<string value expression> ::=

<character value expression>

<character value expression> ::= <concatenation>

| <character factor>

<concatenation> ::= <character value expression> <concatenation operator> <character factor>

<character factor> ::= <character primary>

<character primary> ::= <Expression Primary> | <StringFunctions>

Character Value Expression

Character Value Expression can simply be a Character Factor or it could be a Concatenation.

Character Factor

Character Factor in turn leads to a character Primary.

Concatenation

Concatenation is a join of character value expression and character factor through a concatenation operator.

Character Primary

Character Primary can be any valid Expression Primary or a string value function.

Example 1

Select 'Mr. ' || postName as name from Post

is an Example of String Expression.

Result

name
Mr. Principal
Mr. Vice Principal
Mr. Teacher

Example 2

Select DAYNAME (Date '2002-12-24') as "Day Name" from Post

Result

DAYNAME
Tuesday

The above result shows one of the 3 rows.

Expression Primary

An Expression Primary can be a SubQuery, Constant, Column Reference, Multi-Valued Expression or a Parenthesized Expression.

Syntax

<Expression primary> ::=

<Subquery>

<Column Reference>

<Constant>

<Multi-Valued Expression>

<Parenthesized Expression>

SubQuery

A SubQuery is nothing but a Query itself. It can be one of the three types:

- Table SubQuery
- Row SubQuery
- Scalar SubQuery

Degree

Degree refers to the number of records that are resulted from a Query.

Cardinality

Cardinality refers to the number of columns that are selected in a Query.

Table SubQuery

Table SubQuery is the one for which both degree and cardinality can be greater than or equal to 0

Row SubQuery

Row SubQuery is the one for which degree is 1 while cardinality can be greater than or equal to 0.

Scalar SubQuery

Scalar SubQuery is the one for which both degree and cardinality are 1.

Example 1

Select schoolName from School where Exists (Select schoolID, schoolName from Classes where schoolID > 0)

In the above query, the inner query will be an example of Table SubQuery if it returns more than a single record for the studentID.

Result

schoolName
Arthur Morgan School

Example 2

Select eMailAddress from School where Exists (Select Distinct schoolID, schoolName from Classes where schoolID > 0)

In the above query, inner query will be an example of Row SubQuery, if it returns a single record.

Result

EmailAddress
info@arthuormorganschool.org

Example 3

Select PhoneNumber from School where Exists (Select Distinct schoolID from Classes where schoolID > 0)

In the above query, inner query will be an example of Scalar SubQuery, if it returns a single record for the studentID.

Result

PhoneNumber
1-828-875-4262

Column Reference

Column Reference can be a single column or a group of column separated by period. It can be a valid identifier or period separated identifier.

Syntax

<column reference> ::= <identifier> [{ <period> <identifier> }...]

<identifier> ::= <regular identifier> | <delimited identifier>

Regular identifier

It is an identifier conforming to the rules of the Identifier specified in SQL-99.

Delimited identifier

Delimited identifier is an identifier enclosed within double quotes. Any word, including keywords, can be a delimited identifier.

Example 1

Select postID from post

In the above example postID is an example of ColumnReference

Result

PostID
1
2
3

Example 2

Select a.postID from post a

In the above example a.postId is also an example of Column Reference.

“dshjh^&*%” or “**SELECT**” is an example of valid delimited identifier.

Result

PostID
1
2
3

Constant

A constant is any valid SQL literal, whose value in the current database instance does not change with time.

Syntax

<constant> ::=

<literal>

<exact numeric literal>

<Boolean literal>

<Boolean literal> ::= TRUE

FALSE

<exact numeric literal> ::= <digit>...

<period> <digit> ...

<digit>... <period> <digit>...

<digit>... E <digit>...

Literal

A literal could be any valid character string literal enclosed in ‘’, or any date literal like **DATE** ‘2002-12-24’, or it could also be any valid **TimeStamp** literal like **TIMESTAMP** ‘2002-12-24 09:34:23’.

Boolean Literal

Boolean Literal is a single constant value specifying TRUE or FALSE.

Exact Numeric Literal

An Exact Numeric Literal could be a digit like 0, 22, 333 or any period digit like .2, .33 or <digit period digit> like 22.22.

Multi-Valued Expression

The term Multi-Valued Expression refers to the expression that has more than one valid expression separated by comma.

Syntax

<multi-valued expression> ::=

<left paren> <Expression> [{<comma> <Expression>}...]

A Multi-Valued Expression is group of Expressions separated by comma and enclosed in parenthesis. The Multi-valued expression becomes more significant if we have to compare different expressions. Suppose we have to compare studentName with 'Catherine' and studentID with 3 in that case we can use either

studentName = 'Catherine' and studentID = 3

And using Multi-Valued expression we can also write the above condition as follows

(studentName, studentID) = ('Catherine, 3')

Examples

Select * from student where (studentID, StudentName) = (2,'John')

Here in the above query

(studentID, StudentName) = (2,'John')

is an example of Multi-Valued expression.

Result

StudentID	StudentName	RollNumber	Gender	StudentAddress	PhoneNumber	ClassID
2	John	1001	M	Oroville City Palace 1735 Montgomery Street Oro...	(408)615-7297	1

Parenthesized Expression

A Parenthesized Expression is simply an expression enclosed with left and right parenthesis.

Syntax

<Parenthesized Expression> ::= <left paren> <Expression> <right paren>

Example

(a * b + c / d) is a valid parenthesized expression.

Predicate

A predicate is an SQL expression that is used in the evaluation of a search condition that is either TRUE or FALSE. TRUE indicates that the expression is correct. FALSE indicates that the expression is incorrect. All SQL values used in a predicate must be of a compatible data type (family) for comparison.

Syntax:

<predicate> ::=

<comparison predicate>

| <between predicate>

| <in predicate>

| <like predicate>

| <null predicate>

| <quantified comparison predicate>

| <exists predicate>

Predicates is the term that collectively refers to the one amongst COMPARISON predicate, BETWEEN predicate, IN predicate, LIKE predicate, NULL predicate, QUANTIFIED COMPARISON predicate, EXISTS predicate.

Multi-Valued Expression is a unique feature provided by Daffodil DB. In Multi-Valued Expression, we can combine multiple expressions separated by a comma. Use of a Multi-Valued Expression in case of predicates allows us to combine two different conditions joined with 'AND', that is to be applied on same operator. The term "Multi-Valued" in Multi-Valued predicates refers that an expression involved in the predicate may be replaced with more than one expression, provided the Cardinality does not mismatch. For example if we have two conditions like

'StudentID = 3 AND SchoolID = 2'

then using Multi-Valued Expression, we can combine the two separate conditions as follows:

(StudentID, SchoolID) = (3, 2)

Examples

Let us take the following examples into consideration that are using those predicates whose results (TRUE, FALSE) are based on the values of the column.

Condition *SchoolID = 1 and StudentID <=2* evaluates to TRUE if SchoolID is 1 and StudentID is less than or equal to 2.

Select School.PhoneNumber, School.EmailAddress, Student.studentID from school, student where SchoolID = 1 and StudentID <= 2.

The evaluation of the above query will return TRUE for all the records for which SchoolID is 1 and Students belonging to that school have StudentID less than or equal to 2.

Result

PhoneNumber	EmailAddress	StudentID
1-828-675-4262	info@arthurmorganschool.org	1
1-828-675-4262	info@arthurmorganschool.org	2

Comparison Predicate

The COMPARISON predicate compares two values and returns TRUE or FALSE depending on whether the two values have been compared successfully or not.

Syntax:

<comparison predicate> ::=

<Expression1> <comp op> <Expression2>

<Expression1> ::= <Expression>

<Expression2> ::= <Expression>

<comp op> ::=

<equals operator>

| <not equals operator>

| <less than operator>

| <greater than operator>

| <less than or equals operator>

| <greater than or equals operator>

Expression

An expression can be one of the following:

LITERAL - quoted string, numeric value, datetime value.

FUNCTION CALL - reference to a built-in SQL function.

SYSTEM VALUE - Current date, Current user.

NUMERIC, BOOLEAN or STRING Expression - Combining Sub Expressions using Operators.

Expression1 and Expression2 further define an Expression. Talking Multi-Valued COMPARISON PREDICATE means an expression of the following type.

(ExpressionA1. . . ExpressionAN) <comp op> (ExpressionB1. . . ExpressionBN)

Here ExpressionA1 will be compared with ExpressionB1, ExpressionA2 will be compared with ExpressionB2, . . . and so on ExpressionAN will be compared with ExpressionBN.

Comparison Operators

The operators referred in the syntax are describes as follows:

Comparison Symbol	Symbol Description	Result Description
=	Equal to	Returns TRUE, if both the values are same.
<>	Not Equal to	Returns TRUE, if first value is not equal to the second value
<	Less Than	Returns TRUE, if first value is less than the second value.
>	Greater Than	Returns TRUE, if first value is greater than the second value.
<=	Less Than or Equal to	Returns TRUE, if first value is less than or equal to the second value
>=	Greater than or Equal to	Returns TRUE, if first value is greater than or equal to the second value.

Example

Let us take the following examples that are using comparison predicates where results (TRUE, FALSE) are based on the values of the column,

SchoolID = 1 and StudentID <= 2 evaluates to TRUE if SchoolID is 1 and StudentID is less than or equal to 2.

Select School.PhoneNumber, Student.StudentID, Student.StudentName, ClassID from School, Student where SchoolID = 1 and StudentID <= 2

will return TRUE for all the records for which SchoolID is 1 and StudentID of the Students belonging to that school have value less than or equal to 2.

Result

PhoneNumber	StudentID	StudentName	ClassID
1-828-675-4262	1	Catherine	1
1-828-675-4262	2	John	1

Select School.EmailAddress, Student.StudentID, Student.StudentName from School, Student where (SchoolID, StudentID) = (1, 5)

will return TRUE for all the records for which SchoolID is 1 and StudentID is equal to 5.

Result

EmailAddress	StudentID	StudentName
info@arthurmorganschool.org	5	Wool

Between Predicate

BETWEEN Predicate is used to find all the values that lie between two values. The **BETWEEN** predicate determines if a value is between a range of values.

For example, Expression1 BETWEEN Expression2 AND Expression3 is equivalent to the following search condition.

Expression1 >= Expression 2 AND Expression 1 <= Expression3

Syntax

<between predicate> ::= <Expression1> [NOT] BETWEEN [ASYMMETRIC | SYMMETRIC] <Expression2> AND <Expression3>

<Expression1> ::= <Expression>

<Expression2> ::= <Expression>

<Expression3> ::= <Expression>

Expression

As explained above, the expression can be any one of the following:

LITERAL - quoted string, numeric value, date-time value

FUNCTION CALL - Reference to built-in SQL function

SYSTEM VALUE - Current date, Current User

NUMERIC, BOOLEAN or STRING Expression - Combining Sub Expressions using Operators.

Between

BETWEEN operates on three expressions. The expression1's value is checked for comparison in the range of expression2 and expression3.

Symmetric

When **SYMMETRIC** is specified, then the boundary values of the range of expression2 and expression3 are *not* taken into account when the expression1's value is checked for any match in the specified range.

Asymmetric

When **ASYMMETRIC** is specified, then the boundary values of the range of expression2 and expression3 are *also* taken into account when the expression1's value is checked for any match in the specified range.

For both **BETWEEN** and **NOT BETWEEN**, **ASYMMETRIC** is default.

Not Between

Not Between operates again on the three expressions. But, now the expression1's value is checked for comparison outside the range of the expression2 and expression3.

Examples

Lets take the following examples that are using BETWEEN predicates where results depends upon the given range.

Select * from Marksrecord where ExamID BETWEEN 0 AND 15 and StudentID NOT BETWEEN SYMMETRIC 1 AND 11

The above query will select all the records for which ExamID falls between 0 and 15 and for which StudentID does not lie between 1 and 11.

Marks	StudentID	SubjectID	ExamID
98	1	1	1
87	1	2	1
99	1	3	1
78	1	4	1
52	1	5	1
77	1	6	1

Select * from MarksRecord where ExamID BETWEEN ASYMMETRIC 1 AND 15 and StudentID NOT BETWEEN SYMMETRIC 1 AND 11

The above query will select all the records for which ExamID falls between 1 and 15, with 1 and 15 included and for which StudentID does not lies between 1 and 11, with 1 and 11 excluded.

Marks	StudentID	SubjectID	ExamID
98	1	1	1
87	1	2	1
99	1	3	1
78	1	4	1
52	1	5	1
77	1	6	1

Like Predicate

In order to search or match any string pattern, **LIKE** predicate is used. The **LIKE** predicate finds a string to determine if the string has that particular pattern. The pattern is a string with a combination of the following special characters: underscore character '_' and percent sign, '%'.

Syntax

<like predicate> ::=

<character match value> [NOT] LIKE <character pattern>

<character match value> ::= <string expression>

<character pattern> ::= <string expression>

Character Match Value

Character Match Value is a string that will be searched to determine if the specified Character Pattern can be found.

Character Pattern

Character pattern is the defined pattern, which actually serves as the criteria against which the search is made.

Like | Not Like

LIKE and **NOT LIKE** are the key words to execute search. Operate on *Character Match Value* and *Character Pattern*. **LIKE** searches for the records matching the specified pattern where as **NOT LIKE** searches for the all the records not matching the specified pattern. **LIKE** predicate is case sensitive.

Example 1

Select studentID from Student where studentName LIKE '_o_n'

On executing the above query, all studentIDs will be selected from Students where second character in the name is 'o' and the fourth character is 'n' and total length of the string will be 4. (like 'John')

Result

StudentID
2
4

Example 2

Select studentName from student where studentName like '%a%e'

On executing the above query, all studentNames will be selected from Students that contain 'a' and ends with 'e' like 'Catherine', 'Cathe', and 'Valine'.

Result

StudentName

Catherine

Cathe

Valine

Exists Predicate

This quantified operator verifies the existence of rows. The Boolean result of an **EXISTS** the number of rows returned by the SubQuery determines predicate.

For **EXISTS**, the Boolean result is **TRUE** if SubQuery returns at least one row and **FALSE** if SubQuery does not return any row.

Syntax

<exists predicate> ::= EXISTS <table subquery>

<table subquery> ::= <subquery>

<subquery> ::= <left paren> <query expression> <right paren>

Here **EXISTS** is a keyword that checks the existence of records in the query referred to by <table subquery>.

Table SubQuery

Table SubQuery is a SubQuery listed in the Parent Query and succeeds the keyword **EXISTS**.

EXISTS

As explained above, **EXISTS** returns **TRUE** if the SubQuery returns at least one row, otherwise **FALSE** is returned.

Examples

Select * from Marksrecord where EXISTS (select examID from Exam where examID = 3)

In the above query, if the result of the inner SubQuery comes out with the number of records more than 0 then all EXISTS Predicate will return true and all records of the *Marksrecord* table will be displayed.

Result

Marks	StudentID	SubjectID	ExamID
98	1	1	1
87	1	2	1
99	1	3	1
78	1	4	1
52	1	5	1
....

In Predicate

You can use IN predicate to return a value list or a SubQuery. The **IN** predicate determines if a value is **TRUE** for a list of values. The **NOT IN** predicate also follows the same format as the **IN** predicate.

Syntax

<in predicate> ::= <Expression1> [NOT] IN <in predicate value>

<Expression1> ::= <Expression><in predicate value> ::=

<table subquery>

| <left paren> <in value list> <right paren><table subquery> ::= <subquery>

<in value list> ::= <expression>[{ <comma> <expression> }...]

Expression

As explained before, the expression can be any one of the following:

LITERAL - quoted string, numeric value, date-time value.

FUNCTION CALL - reference to built-in SQL function.

SYSTEM VALUE - Current date, Current user

NUMERIC, BOOLEAN or STRING Expression - Combining Sub Expressions using Operators.

Table SubQuery

Table SubQuery is the SubQuery listed in the Parent Query and succeeds the **IN** predicate.

IN

IN predicate specifies some particular values instead of a broad range (as with **EXISTS** predicate), where value of expression1 has to be matched.

NOT IN

On using **NOT IN**, the expression1 value is searched for a value *not in* the particularly mentioned values by the predicate.

Example 1

Select * from Teacher where postID IN (2,4,5,6,8,9)

Or

Select * from Teacher where postID IN (select postID from Post where postRank = '2')

In the above query, the IN predicate returns TRUE for the record for which *postID* satisfies with any of the following values 2,4,5,6,8,9. Now since only 2 matches with the *postID*, record corresponding to postID = '2' will get selected from the *Teacher*.

Results

EmployeeID	TeacherName	DateofJoining	DateOfBirth	Salary	DepartName	PostID	SchoolId
2	Mr. Brumfield	1997-07-01	1966-11-27	8500	Science	2	1

Example 2

Select TeacherName, EmployeeID, DateOfJoining from Teacher where (Salary, TeacherName) IN (select Salary, TeacherName from Teacher where Salary > 7000 AND Salary < 10000)

The above query finds a list of names, id and date of joining of teachers in the *Teacher* table and then selects all teacher names whose *Salary* is between 7000 and 10000 from the table.

Result

TeacherName	EmployeeID	DateofJoining
Mr. Brumfield	2	1966-07-27

Null Predicate

The **NULL** predicate determines if a column in a selected row contains the SQL value:

NULL.

If column value is **NULL**, then Daffodil DB returns **TRUE**. Following is the syntax for **NULL** predicate.

Syntax

<null predicate> ::=

<Expression> IS [NOT] NULL

NULL

Null predicate Expression **IS NULL** will return **TRUE** only if value of the expression returns **NULL**.

NOT NULL

Null predicate Expression **IS NOT NULL** will return **TRUE** only if value of the expression returns **NOT NULL**.

Example

Select * from ClassProperties where LecturerDescription IS NOT NULL.

Here, the following records will be displayed where the *LecturerDescription* does not contain NULL.

Result

ClassProperties	LecturerDescription	SubjectID	ClassID	TeacherID
1	Prose	2	3	1
2	Prose	2	2	6
3	Prose	2	2	6
4	Botany	1	1	8
5	Zoology	1	2	8

Quantified Comparison Predicate

In QUANTIFIED COMPARISON predicate, the result depends upon another keyword that defines number of records returned by the right hand side of the operator and the value on the left hand side of it.

Syntax

<quantified comparison predicate> ::=

<Expression> <comp op> <quantifier> <table subquery>

<quantifier> ::= ALL | SOME | ANY

Table SubQuery

Table SubQuery is a SubQuery listed in the Parent Query and succeeds the <quantifier> (**ALL**, **SOME**, **ANY**).

Quantifiers

ALL

In case of ALL quantifier, the Quantified comparison predicate will return **TRUE** if and only if all the values of the table SubQuery satisfies the Expression.

SOME

In case of SOME quantifier, the Quantified comparison predicate will return **TRUE** if some of the values of the table SubQuery satisfies the Expression.

ANY

In case of ANY quantifier, the Quantified comparison predicate will return **TRUE** if and only if any of the value of the table SubQuery satisfies the Expression.

Example 1

Select * from ClassProperties where subjectID > ALL (select __rowId from Subject where subjectName <> 'Computers')

All the records from *ClassProperties* will be selected for which *subjectID* is greater than every *__rowId* selected by a SubQuery.

Result

ClassProperties	LecturerDescription	SubjectID	ClassID	TeacherID
20	Fundamentals	6	3	5
21	Memory	6	2	5
22	Basic	6	1	5

Example 2

Select * from ClassProperties where subjectID > SOME (select __rowId from Subject where subjectName <> 'Computers')

All the records from *ClassProperties* will be selected for which *SubjectId* is Greater than *some* of the *subjectID* selected by the SubQuery.

Result

ClassProperties	LecturerDescription	SubjectID	ClassID	TeacherID
1	Prose	2	3	1
2	Prose	2	2	6
3	Prose	2	2	6
4	Botany	1	1	8
5	Zoology	1	2	8
....

The above table shows the first five results that satisfy the query.

Example 3

Select * from ClassProperties where ` owed` ted < ANY (select __rowId from Subject where subjectName <> 'Computers')

All the records from *ClassProperties* will be selected for which *SubjectId* is greater than *any* ` owed selected by a SubQuery.

Result

ClassProperties	LecturerDescription	SubjectID	ClassID	TeacherID
1	Prose	2	3	1
2	Prose	2	2	6
3	Prose	2	2	6
4	Botany	1	1	8
5	Zoology	1	2	8
.....				

Above result shows first five records, which satisfy the given query.

Contains Predicat*

CONTAINS predicate is used to search columns containing character-based data types. This clause can search single word and phrases, words in close proximity to each other, and by-inflexion form of verbs and nouns.

Syntax

```

<contains clause> ::=
CONTAINS <left paren>
<columnorindexname> <comma> <search expression>
<right paren>
<search expression> ::=
<search term>
| <search expression> <vertical bar> <search term>
<search term> ::=
<search factor>
| <search term> <ampersand> <search factor>
<search factor> ::=
[NOT] <search primary>
<search primary> ::=
<text literal>
| <paren search expression>
<paren search expression> ::=
<left paren> <search expression> <right paren>
<text literal> ::=
<word>
| <phrase>

```

columnorindexname - is the name of a specific column or name of a **full-text index**. Name of the index is used when index is created on more than one column.

search expression – is the search criterion containing words, phrases and logical NOT, AND, OR operators.

word - is a string of characters without spaces.

phrase - is one or more words with spaces between each word.

Full-text index can be created by the following syntax:

```

CREATE FULLTEXT INDEX <full-text index name> ON <table name><on column>
<on column> ::= <left paren> <column name list> <right paren>

```

* Features that are not supported in One\$DB
--

full-text index name – user defined full text index name.

table name – name of the table, which is required to be full text enabled.

column name list – name of the column(s) on which full text index is to be created.

Example

The following statement creates a full text index on **StudentAddress** column of *Student* table. The data type of address column is VARCHAR; hence, the full-text index can be created on StudentAddress column.

Create fulltext index school_fulltext on student (StudentAddress)

The full-text search can be applied on the full-text enabled column i.e. StudentAddress, in the following ways:

Example 1. The following query retrieves names of the students living in New York, from the *Student* table.

SELECT StudentName FROM student where contains (StudentAddress,"NY")

Result

StudentName
Cathe
Tovera

Example 2. The following query retrieves names of the students from the *Student* table having Palace or Park in their address field.

SELECT StudentName FROM student where contains (StudentAddress,"Palace" | "Park")

Result

StudentID	StudentName
2	John
3	Cathe
8	Williams

Example 3. The following query retrieves names of the students from the *Student* table living near Airport in Columbia.

SELECT StudentName FROM student where contains (StudentAddress,"AirPort" | "Columbia")

Result

StudentName
Liebig

* Features that are not supported in One\$DB
--

Example 4. The following query retrieves names of the students from the Student table not living in New York and NewZelands.

```
SELECT studentID, StudentName FROM student where contains (StudentAddress, Not ("NJ" | "NY"))
```

Result

StudentID	StudentName
1	Catherine
2	John
4	John
5	Woll
7	Liebig
8	Williams
10	WinkField

* Features that are not supported in One\$DB
--

Data Definition Language

CREATE Database

Use CREATE DATABASE to create a database.

Syntax

```
CREATE DATABASE <database name> [ FILESIZE <equals operator> <large object length> ]  
[ FILEGROWTH <equals operator> <unsigned integer> ]  
[ UNICODE SUPPORT <truth value> ]  
[ USER <user name> PASSWORD <password name> ]  
<large object length> ::= <unsigned integer> | <unsigned integer> <multiplier>  
| <large object length token>  
<truth value> ::= True | False
```

database name

It is the name of the database being created.

FILESIZE <equals operator> <large object length>

This specifies the size of database, in case a new database is created. Default value for this parameter is 5m.

FILEGROWTH <equals operator> <unsigned integer>

This is an integer, which specifies the factor by which database size has to be increased after the space allocated to the database has been taken up or create subsequent file. This is expressed in terms of percentage of the current size of the database. Default value for this parameter is 20%. Valid values for this parameter are 10 to 100.

UNICODE SUPPORT <truth value>

It is used for Multilanguage support.

USER <user name>

The value for this parameter specifies the name of the user creating the database. Default value for this parameter is the current user.

PASSWORD <password name>

The value for this parameter specifies password of the user creating database. Default value for this parameter is password of the current user.

Examples

```
CREATE DATABASE ANSII FILESIZE = 6m FILEGROWTH =12 UNICODE SUPPORT true  
USER ANSII PASSWORD ANSII
```

This will create a database called ANSII for USER ANSII with the FILESIZE = 6m, FILEGROWTH = 12 and UNICODESUPPORT = TRUE.

Note:-

- *IF USERNAME is not specified, then database is automatically created for the current user.*
- *IF FILESIZE is not specified, then Default value of FILESIZE is taken as 5m for the database.*
- *IF FILEGROWTH is not specified, then Default value of FILEGROWTH is taken as 20% for the database.*

DROP Database

To drop a database, use the SQL command DROP DATABASE.

Syntax

```
DROP DATABASE <database name> USER <user name> PASSWORD <password name>
```

Database name

It is the name of the database, which needs to be dropped. For <database name>, any existing database name can be used.

USER <user name >

The value for this parameter specifies the name of the user connecting to the database. There is no default value for this parameter.

PASSWORD <password name>

The value for this parameter specifies user password for connecting to the database. There is no default value for this parameter.

Examples

```
DROP DATABASE ANSII USER ANSII PASSWORD ANSII
```

This will drop the database called ANSII for USER ANSII.

Note: - No default value for USERNAME. If USERNAME is not specified, then an SQL Exception will occur.

Create Table Statement

A **CREATE TABLE** statement creates a table.

A table is a collection of rows having one or more columns. A row is a nonempty sequence of values that corresponds to the column objects in a table. Every row of a table has same number of columns and contains value for each column of the table. Row is the smallest unit of data that can be inserted into a table and deleted from a table.

The **degree** of a table is the number of rows of that table.

Cardinality is defined as the number of columns in a table.

A table whose degree is 0 (zero) is said to be **empty**.

A table is either a base table or a derived table.

Base Table

Base table is the table wherein data is actually stored in the database. All base tables are updatable.

Derived Table

A table obtained from other tables directly or indirectly through the evaluation of a query expression.

Derived tables are either updatable or not updatable. The operations of update and delete are permitted for updatable tables, subject to Access Rule constraints.

Syntax

```
CREATE TABLE <table name> < table element list > [ COUNTRY <country code>
LANGUAGE <language code> ]

<table element list> ::= <left paren> <table element> [ { , <table element> }... ] <right paren>

<table element> ::= <column definition> <table constraint definition>

<column definition> ::= <column name> { <data type> | <domain name> } [ <default clause> ] [
AUTOINCREMENT ] [ <column constraint definition>... ]

<column constraint definition> ::= [ <constraint name definition> ] <column constraint> [
<constraint characteristics> ]

<constraint name definition> ::= CONSTRAINT <constraint name>

<constraint characteristics> ::= <constraint check time> [ [ NOT ] DEFERRABLE ]
|[ NOT ] DEFERRABLE [ <constraint check time> ]

<constraint check time> ::= INITIALLY DEFERRED | INITIALLY IMMEDIATE

<column constraint> ::= NOT NULL | <unique specification> | <references specification> | <check
constraint definition>

<default clause> ::= DEFAULT <default option>

<table constraint definition> ::= [ <constraint name definition> ] <table constraint> [ <constraint
characteristics> ]

<table constraint> ::= <unique constraint definition> <referential constraint definition> <check
constraint definition>
```


<unique constraint definition> ::= <unique specification> <left paren> <unique column list> <right paren> | UNIQUE (VALUE)

<unique specification> ::= UNIQUE | PRIMARY KEY

<unique column list> ::= <column name list>

<referential constraint definition> ::= FOREIGN KEY <left paren> <column name list> <right paren> <references specification>

<references specification> ::= REFERENCES <referenced table and columns> [MATCH <match type>] [<referential triggered action>]

<match type> ::= FULL | PARTIAL | SIMPLE

<referenced table and columns> ::= <table name> [<left paren> <column name list> <left paren>]

<reference column list> ::= <column name list>

<referential triggered action> ::= <update rule> [<delete rule>] | <delete rule> [<update rule>]

<update rule> ::= ON UPDATE <referential action>

<delete rule> ::= ON DELETE <referential action>

<referential action> ::= CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION

<check constraint definition> ::= CHECK <left paren> <search condition> <right paren>

TableName

TableName is the name of the table structure.

Country and Language

It is optional. User may specify the country code and language in which he/she wants to store the data. All string type data will be stored in the user specified language using Unicode.

Table Element List

It refers to a list of different table elements.

Table Element

Each table element consists of Column Definition and Table Constraint Definition

Column Definition

The **column definition** contains all the information needed to define the columns that are part of a table.

This includes:

Column Name

Column name is the name of the column structure within a table created with **CREATE TABLE** statement. Column names must conform to the rules for identifiers and it must be unique in a table.

Data type

The data type describes the type of data that can be stored in the column.

Domain name

The **domain name** refers to the domain corresponding to a given column.

Default clause

The **default clause** allows one to specify default values for a given column. Possible default values can be any literal/null value/datetime value function/ USER /CURRENT_USER/CURRENT_ROLE/SESSION_USER/CURRENT_PATH or any implicitly typed value specification.

Auto increment

Auto increment Declare the column as auto incremented. By default, column value starts from 1 with an Increment factor = 1.

User can declare only the following data type field as an auto incremented:

BIGINT, BYTE, INT, INTEGER, LONG, SMALLINT, TINYINT, DOUBLE PRECISION, FLOAT, REAL, DEC, DECIMAL, NUMERIC.

Column constraint definition

Different constituents of column constraint definition are as follows:

Constraint Name

Constraint name refers to the identifier for a given constraint.

The column constraint

The column constraint consists is one or more keywords, which restrict the data that can be written to a particular column. The column constraints supported by Daffodil DB are as follows:

- NOT NULL
- PRIMARYKEY
- UNIQUE
- FOREIGN KEY
- CHECK

All column constraints are optional.

Not Null

It indicates that a particular column must have a non-NULL value associated with it.

Primary key

It creates an index for the column. The **PRIMARY KEY** column constraint can specify a single column only. In order to specify a **PRIMARY KEY** constraint with multiple columns, **table constraint** is used.

Unique

It defines a unique key on the column. All values for this column must be unique. The **UNIQUE** column constraint can specify a single column only. In order to specify a **UNIQUE** constraint with multiple columns, **table constraint** is used.

Foreign key

It indicates that a relationship exists between column value of this table (known as the child table) and primary key of the parent table, which is referenced in the **REFERENCES** clause.

Check

The optional **CHECK** keyword indicates that the value of a column to be inserted or updated must meet the criteria of the check constraint.

Constraint characteristics

It defines type of the constraint along with the check time. The constraint can be either initially or immediate/initially deferred.

Immediate constraints

These constraints are applied when the operation is performed on the table.

Deferred

These constraints are applied at the time of commit.

The Table Constraint definition

It allows you to define a constraint that is applicable to the table. Usually this type of constraint is used when you specify multiple columns for any type of constraint. The different constituents of table constraint definition are:

Constraint name definition

This name is used to identify a constraint. Each constraint name must be unique for a table.

Table constraint It can be of any of these types.

Unique constraint The unique constraint defines an explicitly named primary key or unique constraint of one or more columns.

Referential constraint

The referential constraint defines an explicitly named foreign key constraint of one or more columns.

A given **foreign key** and its matching candidate key must contain the same number of columns, N, such as: the Ith column of the foreign key corresponds to the Ith column of the matching key (I = 1 to N), and corresponding columns must have exactly the same data type. The referenced table must have a unique or primary index on the specified columns.

Match Types**MATCH FULL**

Match Full specifies that for each row R1 of the referencing table, either the value of every referencing column in R1 shall be a null value, or the value of every referencing column in R1 shall not be null and there shall be some row R2 of the referenced table such that the value of each referencing column in R1 is equal to the value of the corresponding referenced column in R2.

MATCH PARTIAL

Match Partial specifies that for each row R1 of the referencing table, there shall be some row R2 of the referenced table such that the value of each referencing column in R1 is either null or is equal to the value of the corresponding referenced column in R2.

The referencing table may be the same table as the one referenced.

MATCH SIMPLE

Match Simple is the default type and specifies that for each row R1 of the referencing table, either at least one of the values of the referencing columns in R1 shall be a null value, or the value of each referencing column in R1 shall be equal to the value of the corresponding referenced column in some row of the referenced table.

If **MATCH FULL** or **MATCH PARTIAL** is specified for a referential constraint and if the referencing table has only one column specified in <referential constraint definition> for that referential constraint, or if the referencing table has more than one specified column for that <referential constraint definition>, but none of those columns are Nullable, then the effect is the same as if no <match option> were specified.

Referential Triggered action

It consists of update/delete rule along with the referential action. The referential action can be any of the following:

On delete clause

The **ON DELETE** clause defines the rules for deleting specific columns on the specified table.

On update clause

The **ON UPDATE** clause defines the rules for updating specific columns on the specified table.

If the **ON DELETE** or **ON UPDATE** clauses are omitted, the default is **NO ACTION**.

Referential action

With column constraints you must specify at least one identifier. These are:

CASCADE- has the effect of dropping all SQL objects that are dependent on a particular object.

SET NULL- assigns null values to all components of the target column.

SET DEFAULT- assigns default values to all the components of the target column.

RESTRICT- takes care of what objects are dependent on the object being dropped and if there are dependent objects, then the dropping of the object does not take place.

NO ACTION omits the **ON DELETE** clause.

Check constraint

The **check constraint** defines an explicitly named check constraint of one or more columns.

Examples

Without constraints:

CREATE TABLE Post (PostID INT, PostName VARCHAR (20), PostRank VARCHAR (20))

With column constraints:

```
CREATE TABLE Classes( ClassID int CONSTRAINT Class_PK PRIMARY KEY, ClassName
varchar(20) CONSTRAINT CLASS_NAME_NOT_NULL NOT NULL, SchoolID int
CONSTRAINT Class_School_FK REFERENCES School(SchoolID))
```

With table constraints

```
CREATE TABLE ClassProperties( ClassPropertiesID int CONSTRAINT ClassProperties_PK
PRIMARY KEY,
LecturerDescription VARCHAR(20), SubjectID int, ClassID int, TeacherID int, CONSTRAINT
ClassProperties_Subject_FK FOREIGN KEY (SubjectID) REFERENCES Subject(SubjectID),
CONSTRAINT ClassProperties_Class_FK FOREIGN KEY(ClassID) REFERENCES
Classes(ClassID),
CONSTRAINT ClassProperties_Teacher_FK FOREIGN KEY(TeacherID) REFERENCES
Teacher(EmployeeID))
```

With Referential Triggered action and Match Type:

```
CREATE TABLE MarksRecord( Marks int, StudentID int, SubjectID int, ExamID int,
CONSTRAINT MarksRecord_Subject_FK FOREIGN KEY(SubjectID) REFERENCES
Subject(SubjectID),
CONSTRAINT MarksRecord_Exam_FK FOREIGN KEY(ExamID) REFERENCES
Exam(ExamID),
CONSTRAINT MarksRecord_Student_FK FOREIGN KEY(StudentID) REFERENCES
student(StudentID) MATCH SIMPLE ON DELETE CASCADE)
```

With Country code option

```
CREATE TABLE Post (PostID INT, PostName VARCHAR (20), PostRank VARCHAR (20))
COUNTRY JP LANGUAGE JA
```

This will store above defined table in Japanese language.

CREATE Sequence

To create a sequence, use the SQL command CREATE SEQUENCE.

Syntax

```
CREATE SEQUENCE <local or schema qualified name>
[<initialize sequence> [<initialize sequence>...]]
```

<initialize sequence>:-

```
    <sequence starter> <suinteger>
    | <sequence type>
```

<sequence starter>:-

```
    INCREMENT BY
    | START WITH
```

<sequence type>:-

```
    <maxvalue sequence>
    | <minvalue sequence>
```

| <cycle in sequence>

<maxvalue sequence>:-
MAXVALUE <suinteger>
| NOMAXVALUE
<minvalue sequence>:-
MINVALUE <suinteger>
| NOMINVALUE
<cycle in sequence> ::=
CYCLE
| NOCYCLE

<local or schema qualified name>

It is the name of the sequence so created.

sequence starter

INCREMENT BY

This clause specifies the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. The absolute of this value must be less than the difference of MAXVALUE and MINVALUE. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends. If you omit this clause, the default value of the interval becomes 1.

START WITH

This clause specifies the first sequence number to be generated. Use this clause to start an ascending sequence at a value greater than its minimum or to start a descending sequence at a value less than its maximum. For ascending sequences, the default value is the minimum value of the sequence. For descending sequences, the default value is the maximum value of the sequence.

Note: - This value is not necessarily the value to which an ascending cycling sequence cycles after reaching its maximum or minimum value.

MAXVALUE SEQUENCE

MAXVALUE

It specifies the maximum value, the sequence can generate. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE.

NOMAXVALUE

It indicates a maximum value of 9223372036854775807. This is the default value.

MINVALUE SEQUENCE

MINVALUE

It specifies the minimum value of the sequence. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.

NOMINVALUE

It indicates a minimum value of -9223372036857447808. The default value is 1 .

CYCLE IN SEQUENCE

CYCLE

Specifying CYCLE will indicate that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum value.

NOCYCLE

Specifying NOCYCLE will indicate that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default value.

Example

```
CREATE SEQUENCE orders_seq  
START WITH 1000  
INCREMENT BY 1  
NOCYCLE
```

This will create the sequence orders_seq in the default schema. This sequence provides numbers when *orders_seq.NEXTVAL* or *orders_seq.CURRENTVAL* is called from any statement. The first reference to orders_seq.nextval returns 1000. The second returns 1001. Each subsequent reference will return a value 1 greater than the previous reference.

If you specify none of the following clauses, you create an ascending sequence that starts with 1 and increases by 1 with default upper limit. Specifying INCREMENT BY -1 only creates a descending sequence that starts with Max value.

- To create a sequence that increment without bound, for ascending sequences, omit the MAXVALUE parameter or specify NOMAXVALUE. For descending sequences, omit the MINVALUE parameter or specify the NOMINVALUE.
- To create a sequence that stops at a predefined limit, for an ascending sequence, specify a value for the MAXVALUE parameter. For a descending sequence, specify a value for the MINVALUE parameter. Also specify the NOCYCLE. Any attempt to generate a sequence number once the sequence has reached its limit results in an error.
- To create a sequence that restarts after reaching a predefined limit, specify values for both the MAXVALUE and MINVALUE parameters. Also specify the CYCLE. If you do not specify MINVALUE, then it defaults value is 1).

ALTER Sequence

To alter a sequence use the SQL command ALTER SEQUENCE.

Syntax

```
ALTER SEQUENCE <local or schema qualified name>  
<sequence incrementer> [<sequence incrementer>...]
```

```
<sequence incrementer>::= INCREMENT BY <suinteger> | <sequence type>
```

```
<sequence type>:-
```

```
<maxvalue sequence>
```

```
| <minvalue sequence>
```

```
| <cycle in sequence>
```

| <sequence order>

<maxvalue sequence>:-
MAXVALUE <suinteger>
| NOMAXVALUE
<minvalue sequence>:-
MINVALUE <suinteger>
| NOMINVALUE
<cycle in sequence> ::=
CYCLE
| NOCYCLE

local or schema qualified name

It is the name of the sequence to be altered.

SEQUENCE INCREMENT

It specifies the interval between sequence numbers. This integer value can be any positive or negative integer, but it cannot be 0. The absolute of this value must be less than the difference of MAXVALUE and MINVALUE. If this value is negative, then the sequence descends. If the increment is positive, then the sequence ascends.

MAXVALUE SEQUENCE

MAXVALUE

It specifies the maximum value, the sequence can generate.. MAXVALUE must be equal to or greater than START WITH and must be greater than MINVALUE.

NOMAXVALUE

Specifying NOMAXVALUE indicates a maximum value of 9223372036854775807.

MINVALUE SEQUENCE

MINVALUE

It specifies the minimum value of the sequence. MINVALUE must be less than or equal to START WITH and must be less than MAXVALUE.

NOMINVALUE

Specifying NOMINVALUE indicates a minimum value of -9223372036857447808.

CYCLE IN SEQUENCE

CYCLE

Specifying CYCLE indicates that the sequence continues to generate values after reaching either its maximum or minimum value. After an ascending sequence reaches its maximum value, it generates its minimum value. After a descending sequence reaches its minimum, it generates its maximum.

NOCYCLE

Specifying NOCYCLE indicates that the sequence cannot generate more values after reaching its maximum or minimum value. This is the default value.

Example

```
ALTER SEQUENCE orders_seq  
INCREMENT BY 2  
CYCLE
```

This will alter the sequence orders_seq in the default schema.

DROP Sequence

To drop a sequence use the SQL command DROP SEQUENCE.

Syntax

```
DROP SEQUENCE <local or schema qualified name>
```

<local or schema qualified name>

It is the name of the sequence to be dropped. For <local or schema qualified name> you may use an existing sequence name.

Example

```
DROP SEQUENCE orders_seq
```

This will drop the sequence orders_seq in the default schema.

Create Trigger Statement

Create Trigger statement is used to create a trigger.

A trigger can specify additional constraints and business rules within the database to manage a number of executions of an application. Triggers help us to enforce data integrity rules with actions such as cascading deletes or updates. Triggers can also perform a variety of functions such as issuing alerts, updating other tables, sending e-mails, and other useful actions.

Trigger

Trigger defines a set of actions that are executed when a database event occurs on a specified table. A *database event* can be delete, insert, or update operation that is performed by the user.

Syntax:

```
<trigger definition> ::=
```

```
CREATE TRIGGER <trigger name> <trigger action time> <trigger event> ON <table name> [  
REFERENCING <old or new values alias list> ] <triggered action>
```

```
<trigger action time> ::=
```

```
BEFORE| AFTER
```

```
<trigger event> ::=
```

```
INSERT | DELETE | UPDATE [ OF <column name list> ]
```

```
<triggered action> ::=
```

```
[ FOR EACH { <action type rule> } ] [ WHEN <left paren> <Boolean Expression> <right paren>  
] <triggered SQL statement>
```

```
<action type rule> ::=  
ROW | STATEMENT  
<triggered SQL statement> ::=  
<SQL procedure statement> | BEGIN ATOMIC { <SQL procedure statement> <semicolon> } ...  
END
```

```
<old or new values alias list> ::= <old or new values alias> ...  
<old or new values alias> ::= OLD [ ROW ] [ AS ] <identifier>  
| NEW [ ROW ] [ AS ] <identifier>  
| OLD TABLE [ AS ] <identifier>  
| NEW TABLE [ AS ] <identifier>
```

Trigger name

It defines a unique trigger in a schema.

Trigger action time

It signifies when a trigger can be fired or executed relative to the *trigger event*. It takes one of the following values:

BEFORE

Before indicates the state of database instance at a particular time before the statement's changes are applied and before any constraints had been applied to the target table.

AFTER

After indicates the state of database instance at a particular time after all constraints have been satisfied and after the changes have been applied to the target table.

Trigger event

It *specifies* what type of SQL statement fires or executes the trigger. It can take one of the following SQL statements: **DELETE**, **INSERT**, or **UPDATE**.

Table name

It *specifies* the name of the table to which the trigger belongs. A table is allowed to have multiple triggers.

Referencing clause

This clause defines correlation or alias names for old and new values of a row or for the old or new table. You can use the correlation or alias names in the **WHEN** (*search condition*) clause or in SQL statements of the **trigger body**.

Action Type Rule

This clause determines the number of times that a trigger will be fired for each *triggering event*. It can take the value of either **ROW** or **STATEMENT**.

Statement specifies the *trigger body* to execute only once regardless of the number of rows being modified (deleted, inserted, or updated) by the *triggering event* statement.

Row specifies the *trigger body* to execute once for each row that is being modified by the *triggering event* statement.

When (Boolean Expression) specifies the conditions for executing *triggered SQL statement*. All supported SQL search conditions are allowed in this clause.

Triggered SQL Statement

It specifies an SQL statement that the trigger executes.

Single Statement Execution

You can only use one SQL statement in the *triggered SQL statement*. If the *granularity* of the trigger is ROW or STATEMENT, the *triggered SQL Statement* can only take the values of the following SQL statements.

- Insert
- Update
- Delete

Multiple Statement Execution

However, if you use the BEGIN ATOMIC...END keywords, you can execute any number of statements.

You can define any number of triggers for a single table, including multiple triggers on the same table for the same event. You can create a trigger in any schema except *System.definitions_schema*, which is the system table schema.

Order of Execution

When a database event occurs that fires a trigger, Daffodil DB performs actions in the following order:

- It performs the insert, update, or delete.
- It fires *before* triggers.
- It performs constraint checking (primary key, unique key, foreign key, check).
- It gets fired *after* triggers.

When multiple triggers are defined for the same database event for the same table and for the same trigger time (before or after), then they are fired in the order in which they were created.

Trigger Recursion

It is possible for one trigger to cause itself to fire, and thus it is possible for triggers to recur infinitely.

Examples

Trigger Before Insert at statement level

Create trigger abc before insert on classes for each statement update classproperties set subjectID=1

This trigger is fired before the insertion occurs in the *classes* table and updates the *classproperties* table.

Trigger After Insert with Aliases

```
create trigger abc1 after insert on Teacher REFERENCING NEW DEK for each row update Post
set Postname = 'Teacher' where postID=DEK.PostId
```

This trigger is fired after the insertion occurs in the *Teacher* table and updates the *Post* table. The new inserted row in the *Teacher* table is referenced with the alias DEK.

Multiple Statement Execution In Trigger After Insert With Begin Atomic.. End

```
create trigger abc2 after insert on classes referencing new as newrow for each row BEGIN
Atomic update classes set ClassName='6th' where schoolID = newrow. SchoolId; delete from
classes where ClassName='7th'; end
```

The trigger is fired after insert operation occurs on the *Classes* table.

The sql statement to be executed when a trigger is fired is enclosed within begin, atomic and end keywords.

Create Procedure Statement

SQL Stored Procedure is nothing more than a collection of statement that is executed automatically one after the other by Daffodil DB database server. The *invocation* of a stored procedure is treated as a regular external call. The application waits for the stored procedure to terminate, and parameters can be passed back and forth. Stored procedures can be called locally (on the same system where the application runs) and remotely on a different system. However, stored procedures are particularly useful in a distributed environment since they may considerably improve the performance of distributed applications by reducing the traffic of information across the communication network.

For example, if a client application needs to perform several database operations on a remote server, you can choose between issuing many different database requests from the client site and calling a stored procedure. In the first case, you start a dialog with the remote system every time you issue a request. If you call a stored procedure instead, only the call request and the parameters flow on the line. In addition, the server system executes some of the logic of your application with potential performance benefits at the client site.

Syntax

```
<SQL-invoked procedure> ::= CREATE PROCEDURE <routine name> <SQL parameter
declaration list> <routine characteristics> <routine body>
```

```
<SQL parameter declaration list> ::= <left paren> [ <SQL parameter declaration> [ { <comma>
<SQL parameter declaration> }... ] ] <right paren>
```

```
<SQL parameter declaration> ::= [ <parameter mode> ] [ <SQL parameter name> ] <parameter
type>
```

```
<parameter mode> ::= IN
```

```
| OUT
```

```
| INOUT
```

`<routine characteristics> ::= [<routine characteristic>...]`
`<routine characteristic> ::= <language clause>`
`| SPECIFIC <specific name>`
`<language clause> ::= SQL`
`<routine body> ::= <SQL routine body>`
`| <external java reference>`
`<SQL routine body> ::= <SQL procedure statement>`

`<SQL procedure statement> ::= <SQL Statements>`
`<external java reference> ::= EXTERNAL NAME <external method name> [<java method signature>]`
`<external method name> ::= [{< jar name> <colon>}] <java method name>`
`<Jar name> ::= < catalog id> <period> <schema id> <period> <jar id>`
`| <schema id> <period> <jar id>`
`| <jar id>`
`<Catalog id> ::= <identifier>`
`<Schema id> ::= <identifier>`
`<Jar id> ::= <identifier>`
`<Java method name> ::= <java class name> <double colon> <method identifier>`
`<Java class name> ::= <package identifier> [{<period> <package identifier>}...]`
`<Package identifier> ::= <java identifier>`
`<Class identifier> ::= <java identifier>`
`<method identifier> ::= <java identifier>`
`<java identifier> ::= <identifier>`
`<java method signature> ::= <left paren> [<java parameters>] <right paren>`
`<java parameters> ::= <java datatype> [{ <comma> <java datatype> }...]`
`<java datatype> ::=`
`BIGINT | BINARY | BIT | BLOB | BOOLEAN | CHAR | CLOB | DATE | DECIMAL`
`| DOUBLE | FLOAT | INTEGER | NULL | NUMERIC | REAL | SMALLINT | TIME`
`| TIMESTAMP | TINYINT | VARBINARY | VARCHAR | JAVAPARAMETER`

IN, OUT, INOUT Parameters

According to **SQL-99** specification, a parameter defined in a procedure statement could either be **IN**, **OUT** or **INOUT**.

By default it is **IN**.

IN

User has to provide the value of the parameter while calling procedures.

OUT

User can get its value after calling the procedure.

INOUT

User has to provide the value before and can get the value after calling the procedure.

According to **SQL-99** specification in a particular schema, there could be any number of procedures with the same name but with different number of parameters. e.g.

`abc.procedure_name (IN xyz integer)`

and

`abc.procedure_name (IN xyz integer, IN rst integer)` is valid.

So, to differentiate between these procedures, SQL-99 has specific names. A specific name cannot be repeated in a schema i.e. in a schema you cannot have 2 procedures with the same name. Although specific name is optional as per the specification, yet we recommend you to write the specific name as you cannot drop a procedure without a specific name.

Example 1

The following example inserts a row in the *Student* table.

```
CREATE PROCEDURE Student_row_insert (IN varid INT, IN varname varchar
(20), IN
varrollno int, IN vargender char (1), IN varaddr varchar (80), IN varphone varchar
(20), IN
varclasid int) SPECIFIC STUDENT_ROW_INSERT AS BEGIN INSERT INTO
student
VALUES (varid, varname, varrollno, vargender, varaddr, varphone, varclasid); END;
```

Example 2

This example modifies the salary of a teacher.

```
CREATE PROCEDURE Modify_Teacher_Salary (IN varteacherid int, IN varsalary
int)
SPECIFIC MODIFY_TEACHER_SALARY AS BEGIN UPDATE teacher SET salary
=
varsalary WHERE employeeid = varteacherid; END;
```

Example 3

This example retrieves the Marks of the student for the StudentId passed.

```
CREATE PROCEDURE Student_Mark_InOut_Proc (OUT INOUT_PARAM  
INTEGER)  
SPECIFIC Student_Marks_InOut_Proc as BEGIN SELECT Marks into  
INOUT_PARAM from  
MarksRecord WHERE StudentId=INOUT_PARAM; END;
```

External Java Method

External java reference is used to call the method of a java class.

Though jar name is optional, but the jar, which is specified in the class path, can be provided.

Package name is optional. But if specified with the java class then it will search the given java class in the specified package else java class will be looked for in the current package.

The Specified Class Name must have a constructor for the parameter of java, sql.Connection data type; otherwise an Exception will be thrown.

The java data type in the specified function must be a java parameter for each and every Out or InOut parameter mentioned in the Procedure Definition.

The java data type in the specified function must contain same count of parameters as defined in the Procedure Definition.

The specified Class must contain specified functions having parameters of same count and same data type as defined in the Procedure Definition.

Example

This example call a java procedure

```
CREATE PROCEDURE Procedure_Name (OUT var1 int, IN var2 Boolean, INOUT  
var3 int) specific Specfic_Procedure_Name EXTERNAL  
NAME Jar_Name:java.sql.Connection::SQLConnect(Javaparameter, Boolean,  
javaparameter);
```

Example-1

```
CREATE PROCEDURE Procedure_Name() specific Specfic_Procedure_Name  
EXTERNAL NAME Connection::getConnection ()
```

Example-2

```
CREATE PROCEDURE Procedure_Name (IN a int, IN b Boolean) specific  
Specfic_Procedure_Name EXTERNAL NAME Jar_Name:Connection::displayData  
(Integer, Boolean)
```

Example-3

```
CREATE PROCEDURE Procedure_Name() specific Specfic_Procedure_Name  
EXTERNAL NAME com.java.sql.Connection::setUrl()
```

Example-4

```
CREATE PROCEDURE Procedure_Name() specific Specific_Procedure_Name
EXTERNAL NAME Statement::CreateStatement ()
```

Following are the types of statements that can be used in the body of the procedure:

- 1) **Assignment statement:** It is used for assigning values.

Syntax:

```
SET <assignment target> <equals operator> <assignment source>
```

```
<assignment target> ::= <target specification>
```

```
| <modified field reference>
```

```
| <mutator reference>
```

```
<mutator reference> ::= < mutated target specification> <period> <method name>
```

```
<mutated target specification> ::= <target specification>
```

```
| <left paren> <target specification> <right paren>
```

```
| <mutator reference>
```

```
<assignment source> ::= <value expression>
```

```
| <contextually typed source>
```

```
<contextually typed source> ::= <implicitly typed value specification>
```

```
| <contextually typed row value expression>
```

Example:

```
Set a = 1
```

- 2) **Compound statement:** It is used to write a block of statements collectively at one place.

Syntax:

```
<compound statement> ::= [ <beginning label> <colon> ]
```

```
BEGIN [ [ NOT ] ATOMIC ]
```

```
[ <local declaration list> ] [ <local cursor declaration list> ] [ <SQL statement list> ]
```

```
END [ <ending label> ]
```

```
<beginning label> ::= <statement label>
```

```
<local declaration list> ::= <terminated local declaration>...
```

```
<local cursor declaration list> ::= <terminated local cursor declaration>...
```

```
<terminated local cursor declaration> ::= <declare cursor> <semicolon>
```

```
<SQL statement list> ::= <terminated SQL statement>...
```

```
<ending label> ::= <statement label>
```

Example:


```
BEGIN
```

```
    Set a=1;
```

```
    Insert into student values (101,'daisy');
```

```
END
```

- 3) **Case statement:** It is used to perform some action depending on the set of conditions. It is also used to replace *multiple if statements*.

Syntax :

<case statement> ::= <simple case statement> | <searched case statement>

<simple case statement>::=

CASE <simple case operand 1>

<simple case statement when clause>...

[<case statement else clause>]

END CASE

<simple case operand 1> ::= <value expression>

<value expression> ::= <value expression primary>

| <row value constructor> | <value specification>

| <boolean value expression> | <datetime value expression>

| <string value expression> | <numeric value expression>

<simple case statement when clause> ::=

WHEN <simple case operand 2>

THEN <SQL statement list>

<case statement else clause> ::= ELSE <SQL statement list>

<searched case statement> ::=

CASE

<searched case statement when clause>... [<case statement else clause>]

END CASE

Example :

Case rollno

When 101 then insert into student(name) values('daisy');

When 102 then insert into student(name) values('sanya');

Else

insert into student (name) values ('john');

End case;

- 4) **If statement:** It is used to perform some action depending upon a given set of conditions.

Syntax :

<if statement> ::=

IF <search condition>

<if statement then clause> [<if statement else if clause>...] [<if statement else clause>]

END IF

<if statement then clause> ::=THEN <SQL statement list>

<if statement else if clause> ::= ELSEIF <search condition> THEN <SQL statement list>

<if statement else clause> ::=ELSE <SQL statement list>

Example

```
If (rollno=101) then insert into student(name) values('john');
    Elseif (rollno=102) then into student (name) values ('david');
    Else insert into student (name) values ('sanya');
End If;
```

- 5) **Iterate statement:** This statement is used within a loop. When this statement is encountered, control is transferred to the beginning of the loop.

Syntax :

`<iterate statement> ::= ITERATE <statement label>`

Example:

```
create procedure proc4() specific sproc4
as begin
begin
declare aa int;
set aa = 2;
lab :
repeat
    if aa in (22,24,26,28,10) then
        set aa = aa+3;
        iterate lab;
    else
        insert into school(schoolid,schoolname) values(aa,'a');
    end if;

set aa = aa+3;
until aa>=10
end repeat;
end;
end;
```

- 6) **Leave statement:** This statement is used to come out of a loop. When this statement is executed, control is transferred out of the loop.

Syntax:

`<leave statement> ::= LEAVE <statement label>`

Example:

```
Set a=1;
WHILELABEL : While(a<10) do
Set a=a+1;
If (a=5) then
    leave WHILELABEL;
End while;
```

Note: - In the above case, body of while loop will be executed 4 times, after that control will exit of while loop (when a =5, condition will be true)

7) Loop statement: It is used to execute a group of statements repeatedly.

Syntax :

```
<loop statement> ::= [<beginning label> <colon>]
LOOP
<SQL statement list>
END LOOP [<ending label>]
```

Example :

```
LOOP
Set a=1;
Set a=a+1;
END LOOP
```

Above shown is an infinite loop.

8) While statement :It is used to execute a group of statements repeatedly as long as search condition is true.

Syntax:

```
<while statement> ::= [<beginning label> <colon>]
WHILE <search condition> DO
<SQL statement list>
END WHILE [<ending label>]
```

Example:

```
WHILE a < 20 do
    if mod(a,2)=0 then
        insert into Table_Name(one) values(a);
    else
        insert into Table_Name (three) values(a);
    end if ;
    set a = a + 1 ;
End While;
```

- 9) **Repeat statement:** It is used to execute a group of statements repeatedly until a condition becomes true.

Syntax :

```
<repeat statement> ::= [<beginning label> <colon>]  
REPEAT  
  <SQL statement list>  
UNTIL <search condition>  
END REPEAT [ <ending label> ]
```

Example:

```
REPEAT  
  Insert into Table_Name values(0,1);  
  Set a=a+1;  
UNTIL (a>10)  
END REPEAT;
```

Cursor

Operations in a relational database act on a complete set of rows. The set of rows returned by a SELECT statement consists of all the rows that satisfy the conditions in the WHERE clause of the statement. This complete set of rows returned by the statement is known as the Result Set. Applications, especially interactive online applications, cannot always work effectively with the entire Result Set as a unit. These applications need a mechanism to work with one row or a small block of rows at a time. Cursors are a logical extension to Result Sets that let applications work with the Result Set, row by row.

Syntax

```
<declare cursor> ::=  
DECLARE <cursor name> [ <cursor sensitivity> ]  
[ <cursor scrollability> ] CURSOR  
[ <cursor holdability> ]  
[ <cursor returnability> ]  
FOR <cursor specification>
```

```
<cursor name> ::= <local qualified name>
```

```
<cursor sensitivity> ::=  
SENSITIVE  
| INSENSITIVE  
| ASENSITIVE
```

<cursor scrollability> ::=
SCROLL
| NO SCROLL

<cursor holdability> ::=
WITH HOLD
| WITHOUT HOLD

<cursor returnability> ::=
WITH RETURN
| WITHOUT RETURN

<cursor specification> ::=
<query expression> [<updatability clause>]

<updatability clause> ::=
FOR { READ ONLY | UPDATE [OF <column name list>] }

If <cursor sensitivity > is not specified, then ASENSITIVE is implicit, otherwise cursor is sensitive if SENSITIVE is specified, insensitive if INSENSITIVE is specified and asensitive if ASENSITIVE is specified explicitly.

If <cursor scrollability > is not specified, then NO SCROLL is implicit.

If <cursor holdability > is not specified, then WITHOUT HOLD is implicit.

If <cursor returnability > is not specified, then WITHOUT RETURN is implicit.

If <updatability clause> is not specified, then:

- a) If either INSENSITIVE, SCROLL, or ORDER BY is specified, or if *QE* is not a simply updatable table, then an <updatability clause> of READ ONLY is implicit.
- b) Otherwise, an <updatability clause> of FOR UPDATE without a <column name list> is implicit.

If an <updatability clause> of FOR UPDATE with or without a <column name list> is specified, then INSENSITIVE shall not be specified and *QE* shall become updatable.

If an <updatability clause> specifying FOR UPDATE is specified or implicit, then *cursor* is *updatable*, otherwise *cursor* is *not updatable*.

If WITH HOLD is specified, then the cursor specified by the <cursor specification> is said to be a *holdable cursor*.

If WITH RETURN is specified, then the cursor specified by the <cursor specification> is said to be a *result set cursor*.

```
DECLARE num_salary, str_emp_code int;
DECLARE cursoremp cursor for
    select emp_code, salary from employee where deptno = 1;
open cursoremp;
IF cursoremp%ISOPEN THEN
    FETCH cursoremp INTO str_emp_code,num_salary;
    lab : while cursoremp%FOUND do
        UPDATE employee SET salary = num_salary + (num_salary * 0.05) where
            emp_code = str_emp_code;
        insert into emp_raise values (str_emp_code,num_salary * 0.05);
        FETCH cursoremp INTO str_emp_code,num_salary;
    end while;
    close cursoremp;
END IF;
```

Note: - In the example given above, a cursor named cursoremp is used to increment the salary of all employees from department with department code 1 and to insert the incremented amount in another table named emp_raise.

Create View Statement

A Create View Statement creates a view.

View

Derived Tables or “Virtual Tables” are known as Views. They provide an alternative way to look at the data of one or more tables. This virtual table or view derives its values from the evaluation of a query expression in the Create View statement. The query expression can reference base tables, other views, aliases, etc. Essentially, a view is a stored Select statement, of which results can be retrieved at a later time by querying the view as if it was a table. A view can be read-only or updatable.

Syntax

<View definition>:= CREATE VIEW <table name> <regular view specification> AS <query expression>

<regular view specification>:= [<left paren> <view column list> <right paren>]

<view column list>:= <column name list>

Regular View Specification

It specifies a column List, where the names used would be taken as view column Names.

This is optional and follows the rules given below:

If it is not null, then all columns in a column list would be taken as columns of view. If it is null, then column names in a view would be taken from query expression.

Column count in the column list of regular view specification should be equal to the selected column list in the select list of the select query of Query Expression.

Query Expression

It is a select query, where results create view definition. So, if a column list in the regular view specification is null, then selected column list in the select list of a select query would be taken as column names in the view.

Rules for select query are

In the following cases, alias name is essential in the select list of select query

e.g. (a+b), (a*b) etc.

If a column list in regular view specification is not null, then there is no need for an alias name.

Example 1

CREATE VIEW v1 AS SELECT DateOfJoining as Joining Time FROM Teacher

By this query one can create a view having column name as JoiningTime.

Example 2

CREATE VIEW v2 (StudentID, Marks) AS SELECT Student.StudentID, Marks FROM Student, MarksRecord WHERE Student.StudentID = MarksRecord.StudentID

By this view, one can create a view having column names = StudentID and Marks. This view will show the marks of each student along with StudentID. It will take values from tables *Student* and *MarksRecord*.

Create Index Statement

It creates an index on a given table. Only table or view owner can create indexes on that table. The owner of a table can create an index at any time, irrespective of whether there is data in the table or not. Indexes are mainly created to make the retrieval faster in the case of ORDER BY and condition queries referring index column.

Syntax

```
CREATE INDEX <index name> ON <table name> <left paren> <column name> [ ASC | DESC ]  
[ { , <column name> [ ASC | DESC ] }... ] <right paren>
```

index name

It is the name of the index. Index names must be unique within a table but do not need to be unique within a database. Index names must follow the rules of identifiers.

table name

It is name of the table already created that contains column or columns to be indexed. Specifying a catalog name and a schema name is optional.

column

It is the column or columns on which the index is made. Specify two or more column names to create a composite index on the combined values in the specified columns. List the columns to be included in the composite index (in sort-priority order) inside the parenthesis after *table*.

ASC or DESC

Determine the ascending or descending sort direction for the particular index column. The default is **ASC**.

Example 1

```
CREATE INDEX TeacherNameIndex1 ON Teacher (TeacherName)
```

OR

```
CREATE INDEX TeacherNameIndex1 ON Teacher (TeacherName ASC)
```

It creates the index with a name TeacherNameIndex1 on *Teacher* table, which helps in the quick retrieval of data on the column TeacherName, maintaining index in the ascending order.

Example 2

```
CREATE INDEX TeacherNameIndex2 ON Teacher (TeacherName DESC)
```

It creates the index with a name TeacherNameIndex2 on *Teacher* table, which helps in the quick retrieval of data on the column TeacherName, maintaining index in the descending order.

Example 3

```
CREATE INDEX DepartmentTeacherNameIndex ON Teacher (DepartName ASC, TeacherName DESC)
```

It creates the index with a name DepartmentTeacherNameIndex on *Teacher* table, which helps in the quick retrieval of data on the columns Department and TeacherName, maintaining index in the ascending order of Department and the descending order of TeacherName.

Creating too many indexes may increase memory usage and slow down the working of data manipulation commands.

Create FullText Index Statement*

It creates FullText index on a given table. Only the table or view owner can create FullText indexes on that table. Owner of the table can create FullText index at any time irrespective of whether there is data in the table or not. FullText Indexes are mainly created to make the retrieval faster in the case of ORDER BY and condition queries referring FullText indexed column.

Syntax

```
CREATE FULLTEXT INDEX <Index name> ON <table name> <left paren> <column name> [{,  
<column name>}...] <right paren>
```

index name

It is the name of the FullText Index. FullText index names must be unique within a table but does not need to be unique within a database. FullText index names must follow the rules of identifiers.

table name

It is the name of the already created table that contains the column or columns to be indexed. Specifying the catalog name and schema name is optional.

column

It is the column or columns on which the FullText index is made. Specify two or more column names to create a composite FullText index on the combined values in the specified columns. List the columns to be included in the composite FullText index inside the parenthesis after *table*.

Example 1

```
CREATE FULLTEXT INDEX TeacherNameIndex1 ON Teacher (TeacherName)
```

Create FullText index with the name TeacherNameIndex1 on *Teacher* table, which helps in quick retrieval of data on the column TeacherName.

Example 2

```
CREATE FULLTEXT INDEX DepartmentTeacherNameIndex ON Teacher (DepartName,  
TeacherName)
```

Create FullText index on multiple columns with the name DepartmentTeacherNameIndex on *Teacher* table, which helps in quick retrieval of data on the columns DepartmentName and TeacherName.

Creating too many FullText indexes may increase the memory usage and slows down the working of data manipulation commands as well.

* Features that are not supported in One\$DB
--

Create Domain Statement

Create Domain Statement is a domain definition that specifies a data type. It may also specify a <domain constraint> that further restricts the valid values of the domain or a <default clause> that specifies the value to be used in the absence of an explicitly specified value or column default.

Domain

A domain is a set of permissible values. A domain is defined in a schema and is identified by a <domain name>. The purpose of a domain is to constraint the set of valid values that can be stored in a column of a base table by various operations.

Syntax

```
<domain definition> ::= CREATE DOMAIN <domain name> [ AS ] <data type> [ <default clause> ]
```

```
[ <domain constraint>... ]
```

```
<domain constraint> ::= [ <constraint name definition> ] <check constraint definition> [ <constraint characteristics> ]
```

Domain name

It is the name of the domain to be created.

Data Type

The data type description of the data type of the domain.

Default clause

Default Clause is used to define a default value for the domain.

Domain constraints

Domain constraint is a constraint that is specified for a domain. It is applied to all columns that are based on that domain, and to all values directed to that domain.

Example

```
CREATE DOMAIN intdom as Integer check (value > 100)
```

This domain will have integer values with a constraint that the values should be greater than 100.

Create Schema Statement

CREATE SCHEMA statement creates a schema in the database. Schema names must be unique within the database.

Schema

Databases contain collections of independent schemas. A schema is a logical grouping of tables, indexes, triggers, routines, and other data objects under one qualifying name.

User that creates a schema owns that schema unless the optional AUTHORIZATION qualifier is used to specify another user. The schema owner can grant applicable privileges to appropriate users.

Syntax

<schema definition> ::= CREATE SCHEMA <schema name clause> [<schema element>...]

<schema name clause> ::= <schema name> AUTHORIZATION <schema authorization identifier>

AUTHORIZATION <schema authorization identifier>

| <schema name>

<schema authorization identifier> ::= <authorization identifier>

<schema element> ::=

<table definition>

| <view definition>

| <domain definition>

| <trigger definition>

| <schema routine>

| <grant statement>

| <role definition>

| <grant role statement>

Table definition

This is used to create a table.

View definition

This is used to create a view.

Domain definition

This is used to define a domain.

Trigger definition

This is used to create a trigger.

Schema routine

This is used to define a schema procedure or a schema function.

Grant statement

This is used to grant privileges and role authorizations.

Role definition

This is used to create a role.

Examples

Create a schema for sample database

```
CREATE SCHEMA SampleDatabaseSchema
```

Create a table called Post in schema SampleDatabaseSchema

```
CREATE TABLE SampleDatabaseSchema.Post (PostID int, PostName varchar (20), PostRank  
VARCHAR (20))
```

Example 1

```
CREATE SCHEMA Schema_Name AUTHORIZATION User_Name CREATE Role Role_Name
```

Example 2

```
CREATE SCHEMA Schema_Name AUTHORIZATION User_Name CREATE TABLE  
Table_Name ( Column1 integer , Column2 varchar(20) )
```

Create User Statement

Create user Statement is used to create a user in the database. By default, user has no access to SQL data objects like table, schema etc. until the user creates its own tables and schemas or he had been explicitly granted privileges by another user to create data objects.

Syntax

```
CREATE USER <user name> PASSWORD <password name>
```

User name

It specifies the name of the new user. You cannot use the keyword PUBLIC or an existing role name for the user name.

Password

It is the password associated with the user.

The user name and password name must follow the rules of SQL Identifiers.

Example

```
Create User Marty PASSWORD marty
```

Above example creates a user 'Marty' in the database whose password is 'marty'.

```
Create User user1 PASSWORD user1
```

```
Create User user2 PASSWORD user2
```

```
Create User user3 PASSWORD user3
```

Above examples creates multiple users with name 'user1', 'user2', 'user3' in the database whose password is 'user1', 'user2', 'user3'.

Alter Table Statement

Alter table changes the table definition and modifies the structure of the table.

The **ALTER TABLE** statement allows you to:

- Add column to a table.
- Add constraint to a table.
- Drop an existing constraint from a table.
- Add a default value for an existing column in a table.
- Drop a default value for a column in a table by setting the default value to null.

Syntax

ALTER TABLE <table name> <alter table action>

<alter table action> ::= <add column definition> | <alter column definition> | <drop column definition> | <add table constraint definition> <drop table constraint definition>

<add column definition> ::= ADD [COLUMN] <column definition>

<alter column definition> ::= ALTER [COLUMN] <column name> <alter column action>

<alter column action> ::= <set column default clause> <drop column default clause>

<set column default clause> ::= SET <default clause>

<drop column default clause> ::= DROP DEFAULT

<drop column definition> ::= DROP [COLUMN] <column name> <drop behavior>

<add table constraint definition> ::= ADD <table constraint definition>

<drop table constraint definition> ::= DROP CONSTRAINT <constraint name> <drop behavior>

Table name

The *table name* refers to an existing table in the database.

Alter table action

The action allows adding or dropping a constraint or column. It can be of the following types:

Add column definition

This definition adds a column to a table. The syntax for the Column Definition for a new column is same as that for a column in the CREATE TABLE statement. This means that a column constraint can be placed on the new column within the ALTER TABLE ADD COLUMN statement. However, column with a NOT NULL constraint can be added to an existing table if and only if the table is empty; otherwise, an exception is thrown, when the ALTER TABLE statement is executed.

Alter column definition

It changes the column and its definition. The alter column action can be of any of the following types:

- 1) *Set column default clause:* Sets the default clause for a column.

2) *Drop column default clause*: Drops the default clause from a column.

Drop column definition

This definition destroys a column of the base table depending upon the drop behavior.

Add Table Constraint Definition

This definition adds a Constraint to a table.

Drop Table Constraint Definition

This definition destroys a constraint on the table depending upon the drop behavior.

Drop behavior

Drop behavior can be either restrict or cascade.

The optional RESTRICT qualifier to a DROP statement allows a drop only if no objects are dependent on the column or constraint. The optional CASCADE qualifier to a DROP statement drops all related objects to the column or constraint.

Examples**Add a new column with a column-level constraint to an existing table.**

An exception will be thrown if the table contains any rows.

```
ALTER TABLE ClassProperties ADD COLUMN Sections VARCHAR(6) CONSTRAINT  
new_constraint NOT NULL
```

Add a default value to a column (existing rows are not affected).

```
ALTER TABLE Post alter column postrank set DEFAULT '1'
```

Add a table constraint.

```
Alter Table marksRecord Add constraint marksCheck_constraint check(marks<100)
```

Drop a table constraint.

```
Alter table marksRecord Drop constraint marksCheck_constraint cascade
```

Drop Table Statement

DROP TABLE removes the specified table.

Syntax

DROP TABLE <table name> [<drop behavior>]

<drop behavior> ::= CASCADE | RESTRICT

Restrict

If RESTRICT is specified, and if there are any table constraints, or views that use the *table name*, then neither the table is dropped nor the table constraints or the views referring it.

Cascade

With CASCADE, all indexes, columns, constraints, triggers, and SQL routines that are associated with *table name* are dropped as well as the table. RESTRICT is by default.

Example

Drop table Student restrict

This is used to drop the table *Student*.

Drop View Statement

A Drop View Statement drops a view.

Derived Tables or “Virtual Tables” are known as Views. They provide an alternative way to look at the data of one or more tables. This virtual table or view derives its values from the evaluation of a query expression in the Create View statement. The query expression can reference base tables, other views, aliases, etc. Essentially, a view is a stored Select statement, of which you can retrieve results at a later time by querying the view as though it was a table. A view can be read-only or updatable.

Syntax

<drop view statement> ::= DROP VIEW <table name> <drop behavior>

<drop behavior> ::= CASCADE | RESTRICT

CASCADE

If a view is referenced i.e. used by some other objects, then view and objects, which are referring this view are also dropped.

RESTRICT

If a view is referenced or is in use by some other objects, then this view is not dropped.

Example

DROP VIEW v1 RESTRICT

Above example drops view v1, if it is not referenced by any another SQL object.

DROP VIEW v2 CASCADE

Above example drops view v2 and all other SQL object referring this view.

Drop Index Statement

This statement drops the specified index from the database on the specified table. Indexes are also dropped, if you explicitly drop the table on which indexes are created.

Syntax

`DROP INDEX <index name> Of <table name>`

Index Name

It is the name of the index to be deleted.

Table

It is the name of the table on which index to be deleted was created.

Example

`DROP INDEX TeacherNameIndex1 OF Teacher.`

Above Example deletes the index TeacherNameIndex1 on table *Teacher*.

Drop FullText Index Statement*

This statement drops a specified FullText index from the database on a specified table. FullText indexes are also dropped, if you explicitly drop the table on which FullText indexes are created.

Syntax

`DROP FULLTEXT INDEX <index name> of <table name>`

Index Name

It is the name of the FullText index to be deleted.

Table

It is the name of the table on which the FullText index to be deleted was created.

Example

`DROP FULLTEXT INDEX TeacherNameIndex1 OF Teacher.`

Above Example deletes the FullText index TeacherNameIndex1 on the table *Teacher*.

Drop Schema Statement

The DROP schema statement destroys a schema in the database.

Schema

A schema is a logical grouping of tables, indexes, triggers, routines, and other data objects under one qualifying name.

Syntax

`DROP SCHEMA <schema name> <drop behavior>`

`<Drop behavior> ::= CASCADE`

`| RESTRICT`

* Features that are not supported in One\$DB
--

Schema name

The schema name refers to the unique name of the schema.

Drop behavior

If RESTRICT is specified, and if there are any tables or SQL routines or etc. in the *schema name*, then the schema is not dropped and neither are the tables in the SQL routines. With CASCADE, all tables, indexes, columns, constraints, triggers, and SQL routines etc. that are associated with *schema name* are dropped along with the schema.

RESTRICT is by default.

Example

Drop Schema SampleDatabaseSchema cascade

This is used to drop the schema SampleDatabaseSchema.

Drop Procedure Statement

Drop procedure statement is used to drop a previously defined SQL stored procedure. To drop a procedure you should have corresponding specific name of the SQL stored procedure. Because of this it is recommended that you specify a <specific name> at the time of defining SQL stored procedures.

Syntax

DROP <specific routine designator> <drop behavior>

<drop behavior> ::=

RESTRICT

| CASCADE

<specific routine designator> ::= SPECIFIC PROCEDURE <specific name>

CASCADE

Cascade statement drops procedure and all its dependent objects.

RESTRICT

Restrict drops Procedure only if there are no other dependent objects.

Example

DROP SPECIFIC PROCEDURE student_row_insert CASCADE

Above Example drops procedure named *student_row_insert* and all its dependent objects.

Drop Trigger Statement

Drop Trigger Statement removes trigger from the current database. You can remove a trigger by dropping the trigger itself or by dropping the trigger table.

Syntax

DROP TRIGGER <trigger name>

Trigger name

It is the name of the trigger that is to be removed.

When a table is dropped, all triggers on that table are automatically dropped.

Note: - You don't have to drop table triggers before dropping the table.

Example

DROP TRIGGER Insert_Teacher_Trigger

Above Example removes the trigger Insert_Teacher_Trigger entry specified in the System Table.

Drop User Statement

Drop user Statement is used to drop an existing user from database. When you drop an existing user (suppose 'Marty') from database then all the schema objects (like tables, views, procedures etc...) whose owner is user 'Marty' are dropped implicitly.

Syntax

DROP USER <user name>

User name

User name specify an existing user in database.

Example

Drop User Marty

Above example drops an existing user 'Marty' from database.

PSM

PSM stands for Persistent Stored Module. The purpose of PSM is to combine database language and procedural programming language. PSM extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PSM is a block. All PSM programs are made up of blocks, which can be nested within each other. PSM is structured into blocks and can use conditional statements, loops and branches to control program flow. Variables can be scoped so that they are only visible within the block where they are defined. PSM blocks come in three types; these are procedure, triggers and cursors. All of these block types share most PSM features so during this tutorial the features that apply to all block types will be grouped into single subjects. Typically, each block performs a logical action in the program. A block has the following structure:

BEGIN

/ Declarative section: variables, types, and local subprograms. */*
(Statements that make up the block)

DECLARE

/ Executable section: procedural and SQL statements go here. */*
/ This is the only section of the block that is required. */*
(Definition of any variables or objects that are used within the declared block.)

END;

(End of block marker.)

EXAMPLE OF PSM BLOCK IS :

```
create procedure ddbproc() specific s_ddbproc
begin
  declare aa int;
  set aa = 11;
  lab :
  repeat
    if aa=19 then
      set aa = aa+1;
    leave lab;
  else
    insert into student(studentid,studentname) values(aa,0);
    end if;
    set aa = aa+1;
  until aa>=100
  end repeat;
end
```

EXAMPLE FOR NESTED BLOCK IS:

```
create procedure p1(in a int) specific p
as begin
  begin
    declare d int;
    select studentid into d from student where studentid=10;
    insert into teacher(employeeid,teachername) values(d+10,'a');
  begin
    declare d int;
    set d=10;
    set d=a+d;
    insert into student(studentid,studentname) values(d,'a');
  end;
end;
end;
```

THE RULES OF BLOCK STRUCTURE ARE :-

Every unit of PSM must constitute a block. As a minimum there must be the delimiting words BEGIN and END around the executable statements.

SELECT statements within PSM blocks are embedded SQL (an ANSI category). As such they must return one row only. SELECT statements that return no rows or more than one row will generate an error(but not in case of cursor). If you want to deal with groups of rows you must place the returned data into a cursor. The INTO clause is mandatory for SELECT statements within PSM blocks (which are not within a cursor definition), you must store the returned values from a SELECT.

If PSM variables or objects are defined for use in a block then you must also have a DECLARE section.

PSM blocks may be nested, nesting can occur wherever an executable statement could be placed (including the declare section).

Data Manipulation Language

Insert Statement

An INSERT statement creates a row or rows and stores them in the named table. The number of values assigned in an INSERT statement must be the same as the number of specified or implied columns.

Syntax

INSERT INTO <insertion target> <insert columns and source>

<insertion target> ::= <table name> <insert columns and source> ::= <from subquery> <from constructor> <from default>

<from subquery> ::= [<left paren> <insert column list> <right paren>] <query expression>

<from constructor> ::= [<left paren> <insert column list> <right paren>] <contextually typed table value constructor>

<from default> ::= DEFAULT VALUES

<insert column list> ::= <column name list>

<contextually typed table value constructor> ::= VALUES <left paren> <Expression> [{ <comma> <Expression> } ...] <right paren>

Table name

Table name is the name of a table in which the row will be inserted.

Insert Column list

Is a list of one or more columns in which data is to be inserted. Column list must be enclosed in parentheses and delimited by commas. If a column is not in column list, Daffodil DB automatically provides a value for the column if the column has a default value. If column list is specified then the values inserted through the use of constructor or the SubQuery should come in the same order.

Contextually Type Value Constructor

Value Constructor specifies column values to be inserted in a table. There are two ways with which we can specify column values to be inserted in a column. We can specify values for a record of table or we can specify values for multiple records of a table like:

- Values (1, 2, 'sapling'), specify 1, 2 and 'sapling' as the single record values. In this, cardinality (i.e. no of columns) is 3.
- Values ((1, 2, 'sapling'), (2, 3, 'daffodil'), (3, 4, 'transport')), specify values for 3 records. In this, cardinality is 3.

Insert Columns and Source

There are 3 sources from which we can put values in the table through insert query.

From SubQuery

You can use the output from a SubQuery to insert values into the table specified by insertion target. The following **Constraints** are applied

From SubQuery includes Insert Column List as an Optional Rule. It means value of this rule can be null or not null.

If insert column list is **null** then all the columns of the target table acts a target in which values are to be inserted from the select list of columns of select query, provided cardinality of target table (i.e. no of columns in target table) should be same as the cardinality of select query (i.e. no of columns in the select list of select query) and columns descriptors i.e. data type of columns of target table and select query should be the same.

If insert column list is **not null** then all the columns in insert column list acts a target in which values are to be inserted from the select list of columns of select query, provided cardinality of target table (i.e. no of columns in target table) should be same as the cardinality of select query (i.e. no of columns in the select list of select query) and columns descriptors i.e. data type of columns of target table and select query should be the same.

Example

In the example given below we insert values of columns subjectID, examID for studentID = 3. Values returned by the subquery acts as input to the insert to insert new values in the record.

```
INSERT INTO MarksRecord (subjectID, examID) (SELECT subjectID, examID FROM MarksRecord WHERE StudentID = 3)
```

From Constructor

You can use the output from a value constructor to insert values into the table specified by insertion target. The following **Constraints** are applied:

From Constructor includes Insert Column List as an Optional Rule. It means value of this rule can be null or not null.

If insert column list is **null** then all the columns of the target table acts a target in which values are to be inserted from the values in the constructor, provided cardinality of target table (i.e. no of columns in target table) should be same as the cardinality of constructor (i.e. no of columns in the value constructor) and columns descriptors i.e. data type of columns of target table and value constructor should be the same.

If insert column list is **not null** then all the columns in insert column list acts a target in which values are to be inserted from the value constructor, provided cardinality of target table (i.e. no of columns in target table) should be same as the cardinality of value constructor (i.e. no of columns in the value constructor) and columns descriptors i.e. data type of columns of target table and value constructor should be the same.

Example

In the example given below we insert values in SUBJECT table through the use of constructor. There must be one data value for each column in *column list* (if specified) or in the table. The values list must be enclosed in parentheses. If the values in the VALUES list are not in the same order as the columns in the table or do not have a value for each column in the table, *column list* must be used to explicitly specify the column that stores each incoming value.

```
Insert into SUBJECT VALUES(10,'Biology')
```

For inserting values for more than one records at a time you have to use constructor within a constructor.

```
Insert into SUBJECT values(21,'physics'),(31,'chemistry'),(41,'geography')
```

In example given below default values inserted into the SCHOOL table. If default values does not specified for the columns then **NULL** is inserted

Insert into SCHOOL VALUES(100, default, default, default, default)

Update Statement

UPDATE statement is used to modify existing data in a table. Data can be modified in a single row, a group of rows or all the rows in a table. However, an UPDATE statement referencing a table can change the data only in one base table at a time. The UPDATE statement does not affect the row count of a table.

Syntax

UPDATE <target table> SET <set clause list>[WHERE <search condition>]

<target table> ::= [<left paren>] <table name> [<right paren>]

<set clause list> ::= <set clause> [{ <comma> <set clause> }...]

<set clause> ::= <update target> <equals operator> <update source>

<update target> ::= <column name>

<update source> ::= <expression>

<search condition> ::= <boolean expression>

Target Table

Target table is the name of the table to be updated.

Set Clause List

Set Clause List specifies the list of column or variable names to be updated. It specifies a list of attribute value pairs separated by equals' operator.

Update Target

Update Target is a *Column name* that contains the data to be changed. *Column name* must reside in the table specified in the UPDATE clause.

Update source

It can be any valid SQL expression or a column. SQL Expression value or column value acts as input value to be updated in the column.

Example

In the example given below table STUDENT is updated, the condition specified is the StudentID=2 and value of the StudentName will be changed to 'Fleming' if the condition is met.

Update Student set StudentName='Fleming' where StudentID=2

In the example given below 2 columns are updated by the given values if the condition StudentID=1 is met.

Update Student set StudentName= 'Tony', Gender = 'F' where StudentID=1

Delete Statement

Delete statement deletes one or more than one rows from a table depending upon the condition specified by the user in the WHERE clause. If no condition is specified then all the rows in the table are deleted.

Syntax

```
DELETE FROM <target table> [ WHERE <search condition> ]<target table> ::= [<left paren>]
<table name> [<right paren>]
```

```
<search condition> ::= < Boolean Expression>
```

Target table

It is the name of the table from which the rows are to be removed.

WHERE

WHERE clause specifies the conditions used to limit the number of rows that is to be deleted. Where Clause is Optional in Delete Statement, means either it can present or not. If a WHERE clause is not supplied, DELETE removes all the rows from the table, else deletes the rows of table according to condition specified.

Search Condition

It specifies the restricting condition for the rows to be deleted. There is no limit to the number of predicates that can be included in a search condition.

The DELETE statement may fail if it violates a trigger or attempts to remove a row referenced by data in another table with a FOREIGN KEY constraint. If the DELETE removes multiple rows, and any one of the removed rows violates a trigger or constraint, the statement is cancelled, an error is returned, and no rows are removed.

However, an empty table or view cannot be deleted from the database. To delete it from the database, it must be explicitly removed using the DROP TABLE OR DROP VIEW statement.

Examples

Use DELETE with no parameters

This example deletes all rows from the Post table.

```
DELETE FROM Post
```

Use DELETE on a set of rows

This example deletes all rows in which ClassID is less than 6 from the classproperties table.

```
DELETE FROM classproperties where ClassID < 6
```

Use DELETE based on a SubQuery.

This example is used to delete records from Teacher table that is based on a IN predicate. It removes rows from the Teacher table using a sub query that returns the post id related with post name as 'Teacher'.

```
DELETE FROM Teacher WHERE PostID IN (SELECT PostID FROM Post WHERE
PostName = 'Teacher ')
```

Data Query and Control Language

Select Statement

The **SELECT** Statement is a DQL (Data Query Language). It queries the database and retrieves rows from the database, thus allowing the selection of one or many rows or columns from one or many tables.

Syntax

SELECT [<set quantifier>] [<top function>] <select list> <table expression>

<set quantifier> ::= DISTINCT | ALL

<select list> ::= <select sublist> [{ <comma> <select sublist> }...]

<table expression> ::=

<from clause>

[<where clause>]

[<group by clause>]

[<having clause>]

[<order by clause>]

<top function> ::= TOP <left paren> <unsigned integer> <right paren>

Select List

<select sublist> ::= <derived column> | <qualified asterisk>

<qualified asterisk> ::= <asterisk> | <asterisked identifier chain> <period> <asterisk>

<asterisked identifier chain> ::= <asterisked identifier> [{ <period> <asterisked identifier> }...]

<asterisked identifier> ::= <identifier>

<derived column> ::= <Expression> [<as clause>]

<as clause> ::= [AS] <column name>

Distinct/All

These are the optional set quantifiers. **DISTINCT** specifies the discarding of the duplicate records when the two or more records in the selected columns are same. **ALL**, on the contrary returns all the records including the duplicate records.

TOP Function

The **TOP** Function displays the top 'n' records from the result set, where 'n' is the argument passed to the function.

Select list

It is the list of the columns, separated by comma that the user wishes to retrieve. The Select list, can be

- simply an asterisk, '*', to select all the columns of the table in the **FROM** clause.
- specific fields' names to select selected columns.
- some expression using the aggregate functions or some operator, etc.

Table expression

Table Expression lists the source of the tables from which the columns specified in the <select list> are to be retrieved as well as the conditions that are to be applied over them. Table Expression can be formed of the various clauses as indicated i.e. **FROM** clause that is mandatory and **WHERE** clause, **GROUP BY** clause, **HAVING** clause and **ORDER BY** clause that are optional. These clauses will be explained in detail later.

Example 1

Select * from Subject

SubjectID	SubjectName
1	Biology
2	English
3	Mathematics
4	Science
5	Social Studies

The above query lists all the records from the *Subject* table. This is specified by the asterisk, '*', which is used to select all the rows from the table specified. Here the quantifier is by default *ALL*. This means all the records, irrespective of whether there are any redundant records, will be displayed. The above result shows the first 5 rows.

Example 2

To specify explicitly, the listing of all the records, **ALL** can be used as shown, which is otherwise same as the query above.

Select all * from Subject

Example 3

To remove the duplicate records the keyword **DISTINCT** is used as shown below, which will again list all the records of the *Subject* Table, but this time after dropping the duplicates.

Select distinct * from Subject

SubjectID	SubjectName
1	Biology
2	English
3	Mathematics
4	Science
5	Social Studies

Here the result is same, because there are no duplicate columns.

Example 4

The following example results in the listing of the specified columns only.

Select StudentName, RollNumber, Gender from Student

Result

StudentName	RollNumber	Gender
Catherine	1001	F
John	1002	M
Cathe	1003	F
John	1004	M
Woll	1005	F

The above result shows the first five rows only.

Example 5

select TOP (5) (marks*100 /500) as Percentage, StudentID, SubjectID from MarksRecord

Result

Percentage	StudentId	SubjectID
19.6	1	1
17.4	1	2
19.8	1	3
15.6	1	4
10.4	1	5

The above query is another form of the **SELECT** queries, where we have used **TOP** function to list the top 5 students from the list. To show the implementation of the mathematical expression in **SELECT statement**, we have calculated the percentage taking the maximum marks as 500. Also the column aliasing has been used in the query, where the first column has been renamed to *Percentage*.

Example 6

The following query shows another way of column selection. Here all the columns of *Teacher* get selected, but just one column of *Post*.

Select a.TeacherName, a.Salary, b.PostName from Teacher as a, Post as b where a.PostID = b.PostID

TeacherName	Salary	PostName
Mr. Agregado	10000	Principal
Mr. Brumfield	8500	Vice-Principal
Ms. McKelvey	6000	Teacher
Mr. Everett	6000	Teacher
Mr. Verstrepen	6000	Teacher

Here only the first five records of the query result have been shown.

FROM CLAUSE

It specifies the tables, views, derived tables, and joined tables used in **DELETE**, **SELECT**, and **UPDATE** statements. Based on the **SELECT** statement a few Examples involving *FROM* have been already given. As a matter of fact, the **FROM** clause is mandatory in the **SELECT** statement.

Syntax

```

<from clause> ::= FROM <table reference list>
<table reference list> ::= <table reference> [ { <comma> <table reference> }... ]
<table reference> ::= <table primary> | <joined table>
<table primary> ::= <table or query name> [ [ AS ] <correlation name> ] |
<derived table> [ AS ] <correlation name>
<left paren> <table reference> <right paren>
<derived table> ::= <table subquery>
<joined table> ::= <cross join> | <qualified join>
<cross join> ::= <table reference> CROSS JOIN <table reference>
<qualified join> ::= <table reference> [<outer join type> ] JOIN <table reference> <join
condition>
<outer join type> ::= LEFT | RIGHT | FULL
<join condition> ::= ON <search condition>
<search condition> ::= <Boolean Expression>

```

Table Reference

A list of one or more table or view names (separated by commas), from which the data value are to be retrieved. Table reference can be Simple table or Joined table. The *Joined Table* is obtained as a result of any join operations which will be discussed later.

Table Primary

Table Primary is formed of Table Name or the Query Name. Its represents tables, views, queries and joins

Table or query name: includes name of the table from which we want to retrieve the data. We can specify the name of table with its schema name or without schema name. If no schema name is specified then ,the current schema is assumed.

Joined Table

It specifies the intermediate result table that is the result of either equi-join or an outer join. The operators that could be applied are: **CROSS JOIN**, **LEFT OUTER JOIN**, **RIGHT OUTER JOIN**, and **FULL OUTER JOIN**.

Joined Condition

It defines a search condition in which predicates can be combined.

Examples

Some of the examples have already been discussed in the detailing of the SELECT statement.

Select ExamName, MaximumMarks, PassingMarks from Exam

Result

ExamName	MaximumMarks	PassingMarks
Final year	600	250
Half yearly	600	250
Session1	600	250
Session2	600	250

This query, which is the simplest of all, lists the Name of the Exam, Maximum Marks for the exam and the passing marks of the Exam as the output. We can also select more than one table from in the Table Reference List, the example for which is listed next

Select t.TeacherName,t.DateOfJoining,t.salary,p.PostID From Teacher as t, Post as p

Result

TeacherName	DateOfJoining	Salary	PostID
Mr. Agregado	1996-04-17	10000	1
Mr. Brumfield	1997-07-01	8500	1
Ms. McKelvey	1998-09-25	6000	1
Mr. Everett	1998-10-25	6000	1

The result shows the first four rows of the result only.

The above query takes two tables in the from clause, but it is inefficient in the way that it lists out all the records of the tables, i.e. displays the Cartesian Product of the two tables, which is not what the user wants always. Here, though, only the first four records have been shown. The *p.PostID* field shows the Cartesian product being carried out in the operation. This type of operations, involving more than one table, need to be controlled. The next example shows exactly this, by forcing the condition in the *where* clause.

select t.TeacherName, t.DateOfJoining, t.salary, p.PostID

from Teacher as t, Post as p

where t.PostID = p.PostID

Result

TeacherName	DateOfJoining	Salary	PostID
Mr. Agregado	1996-04-17	10000	1
Mr. Brumfield	1997-07-01	8500	2
Ms. McKelvey	1998-09-25	6000	3
Mr. Everett	1998-10-25	6000	3

The above result shows the first 4 rows of the result.

This is the improvised version of the previous query as it limits the output, and displays only those records for which the PostId value of the tables is same. Such types of operations are also called the *JOIN* operations. We can perform such operations using other keywords, like **INNER JOIN**, **LEFT JOIN** etc., which we are going to discuss next.

JOIN OPERATION

As explained briefly in the above discussion,, there is a need to combine two or more tables for the desired output many a times This leads to a correlation among the tables in the from clause. This is exactly what JOIN does i.e., combining two or more tables to produce the expected results. Join is performed whenever there are more than two tables in the from clause and frequently based on the condition mentioned in the where clause. This condition is better known as join condition.

Without the condition, the join will take the form of *Cartesian Product*, resulting in all the possible combinations of the involved tables. e.g. the join of two tables with 3 and 4 records respectively, will result in the output of 12 (i.e. 4 X 3) records.

Cross join

The *Cross Join* logical operator joins each row from the first table with each row from the second table. Thus it is the basic of all the joins and is not very efficient, but still illustrates the table unification characteristic of all the joins.

Example:

Select Classes.*, Exam.ExamID, Exam.ExamName from Classes CROSS JOIN Exam

Result

ClassID	ClassName	SchoolName	ExamID	ExamName
1	6 th	1	1	Final year
1	6 th	1	2	Half yearly
1	6 th	1	3	Session1
1	6 th	1	4	Session2
2	7 th	1	1	Final year

This is the basic query for the cross join. Since the *classes* table consists of 3 entries and the *exam* table consists of 4 entries, the cross join of the two will result in 12 entries, i.e. 4 X 3 entries. But here, only the first five rows have been displayed.

Qualified join:

The qualified join further is categorized into

Left Outer Join/ Left Join.

Right Outer Join/ Right Join.

Full Outer Join/ Full Join.

Left Outer Join/ Left Join:

In this case all the rows of the Left table will appear at least once. The *Left Outer Join* logical operator returns each row that satisfies the join condition between the Left table and the Right

table. In case a row of Left Table does not match with any row of the Right Table, then the corresponding record will still be displayed with Null value in the corresponding columns of the Right Table.

Example

The *left outer join* can be exemplified by the same query. As already mentioned - applying the *left outer join* outputs those values too, from the *table1* which do not match to any value in the *table2*.

Select

StudentName, ClassName, RollNumber

From

Student LEFT OUTER JOIN Classes

on

Student.ClassID = Classes.ClassID

Result

StudentName	ClassName	RollNumber
Catherine	6th	1001
John	6 th	1002
Cathe	6 th	1003
John	6 th	1004
Woll	6 th	1005

Here again, only the first five rows have been displayed.

Right Outer Join/Right Join :

In this case all the rows of the Right table will appear at least once. The *Right Outer Join/Right Join* logical operator returns each row that satisfies the join condition between the Left table and the Right table. In case a row of Right Table does not match with any row of the Left Table, then the corresponding record will still be displayed with Null value in the corresponding columns of the Left Table.

Example:

Same query can be used to exemplify the *Right Outer Join*. But this time all the rows from the second table are displayed at least once in the result. The rows for which there is no match in the first table are displayed only once after being joined with Null value of the first table.

select

StudentName, ClassName, RollNumber

from

Student RIGHT OUTER JOIN Classes

on

Student.ClassID = Classes.ClassID

Result

StudentName	ClassName	RollNumber
Catherine	6th	1001
John	6 th	1002
Cathe	6 th	1003
...
NULL	8 th	NULL

In the result shown above, first three and the last row has been displayed. The last row shows that for the value in the right table, which did not find any corresponding value in the left table; NULL values are assigned for the corresponding left table values.

Full Outer Join/ Full Join:

The *Full Outer Join* logical operator returns each row satisfying the join predicate from the first table joined with each row from the second table. It also returns rows from:

- the first table that had no matches in the second table.
- the second table that had no matches in the first table.

The input that does not contain the matching values is returned as a null value.

Example:

We consider the same query for the *Full Outer Join* as well.

select

StudentName, ClassName, RollNumber

from

Student FULL OUTER JOIN Classes

on

Student.ClassID = Classes.ClassID

Result

StudentName	ClassName	RollNumber
Catherine	6th	1001
John	6 th	1002
Cathe	6 th	1003
...
NULL	8 th	NULL

Here in the above shown result too, the first three and the last row of the result is displayed. The last row shows that for the value in the right table, which did not find any corresponding value in the left table, NULL values are assigned for the corresponding left table values.

This time, all the rows of first table and second table are displayed as the result. The rows in the first table or the second table which have no matching value in the corresponding table are displayed only once, with the other table value being *Null* value, i.e. if there is some row in the first table which has no match in the second table, is shown once in the result, with the corresponding value of the second table being **NULL** and vice versa.

WHERE CLAUSE

A **WHERE** clause is an optional part of a **SELECT** statement, **DELETE** statement, or **UPDATE** statement. But whenever present, it follows the *from* clause and itself is followed by a Conditional Expression. Thus it performs the job of filtering the rows from the tables listed in *from* clause, which satisfy the following condition.

Syntax

<WHERE CLAUSE> ::= WHERE <search condition>

Search Condition

It defines the condition to be met by the rows to be returned. There is no limit to the number of predicates in the search condition. The various operators that may be used in the conditional expressions are "=", "<", "<=", ">=", etc. Apart from these operators, we can have predicates like *IN predicate*, *Between predicate*, *Like Predicate*, and so on, used to form the search condition.

Examples

select TeacherName, DateOfJoining, Salary from Teacher where EmployeeID <=3

Result

TeacherName	DateOfJoining	Salary
Mr. Agregado	1996-04-17	10000
Mr. Brumfield	1997-07-01	8500
Ms. McKelvey	1998-09-25	6000

The above query outputs the *Name of the Teacher*, *Date of Joining*, and, *Salary* from the *Teacher* table for the employees whose *EmployeeID* is less than or equal to 3.

To take another example involving the implication of the Condition on the string valued fields, the query follows:

select StudentName, RollNumber from Student where StudentName like 'Ca%'

Result

StudentName	RollNumber
Catherine	1001
Cathe	1003

The above query lists the Names, Roll Numbers and Address of those students whose Name begins with 'Ca'. Thus, it will result in the output of two records. Here, like is the function that

selects only those records where name begins with 'Ca'. For obtaining specific values, the <field name> of string type, can also be used using the '=' operator just like its use in case of Integer field types.

```
select StudentName, RollNumber, StudentAddress from Student where StudentName = 'Catherine'
```

Result

StudentName	RollNumber	StudentAddress
Catherine	1001	1500 Warb.....

```
select TeacherName,DateOfJoining,Salary from Teacher where Salary IN (8500,10000)
```

Result

TeacherName	DateOfJoining	Salary
Mr. Agregado	1996-04-10	10000
Mr. Brumfield	1997-07-01	8500

Clearly, the above query lists out the records of all the Teachers whose Salary is either 8500, or 10,000. The IN clause will be detailed later.

GROUP BY CLAUSE

A GROUP BY clause, part of a SELECT statement, groups a result into subsets that have matching values for one or more columns. **GROUP BY** clause is optional and follows the **WHERE** clause, and if **WHERE** clause is not present, it follows **FROM** clause. It operates on the rows filtered by the **WHERE** clause. This clause performs the function of grouping the rows based on the common values in the grouping columns. The **GROUP BY** clause restricts the rows of the result set i.e, in each group, no two rows have the same value for the grouping column or columns. NULLs are considered equivalent for grouping purposes.

If several single row columns are in a query, **GROUP BY** returns exactly as many rows as there are distinct sets of values in all the single row columns involved in the query. If these columns have five sets of values, five rows will result.

You typically use a GROUP BY clause in conjunction with an aggregate expression.

Syntax

```
<group by clause> ::= GROUP BY <grouping specification>
```

```
<grouping specification> ::= <grouping set> [ { <comma> <grouping set> }... ]
```

```
<grouping set> ::= <grouping column reference>
```

Example

In the following query the grouping is done by the ClassID. It combines the *Class* table and the *Student* table and counts the number of students in each class by grouping them with their ClassID i.e. the result of the above query displays the number of students in class with ClassID = 1, ClassID = 2 and so on.

```
select Classes.ClassID, COUNT(Student.StudentName) from Student, Classes where
Student.ClassID = Classes.ClassID group by Classes.ClassID
```

The **COUNT** function returns the count of the argument passed.

Result

ClassID	COUNT
1	5
2	5

The following query calculates the average marks of students according to their *StudentID*, i.e., average marks of all the students with STUDENTID = 1, average marks of all the students with STUDENTID = 2, and so on.

```
select StudentID , Avg (Marks) as AVGS from Marksrecord Group By StudentId
```

Result

StudentID	AVGS
1	81.0
2	82.0
3	78.0
4	79.0

And in this way, average marks for the students grouped by their StudentId will be displayed. Here only till the StudentId = 4, have been displayed.

HAVING CLAUSE

The **HAVING** clause specifies a search condition for the grouping and aggregate queries. In *Grouping Queries*, it follows the **GROUP BY** clause. A **HAVING** clause restricts the results of a **GROUP BY** in a **SELECT** statement. In *Aggregate Queries*, **HAVING** follows the **WHERE** clause and if **WHERE** clause is missing, it follows the **FROM** clause..It is usually used in the **GROUP BY** clause.

Like the **WHERE** clause, **HAVING** filters the query result rows. **WHERE** filters the rows from the **FROM** clause and the **HAVING** clause filters the grouped rows or the aggregated rows.

Syntax

```
<having clause> ::= HAVING <search condition>
```

```
<search condition> ::= <Boolean Expression>
```

Examples

```
select DateOfJoining, count(*) from Teacher Group By DateOfJoining, salary HAVING Salary <=85000
```

Result

DateOfJoining	COUNT
1996-04-17	1
1997-07-01	1

The above query first groups the result set by *DateOfJoining* and salary and then imposes the condition of SALARY <= 8500, thus further filtering the final record set. The above result contains just the first 2 rows of the result set.

UNION OPERATOR

The UNION operator derives a result table by combining two other result tables (for example TABLE1 and TABLE2) and eliminating any duplicate rows in the tables. When **ALL** is used with UNION (that is, **UNION ALL**), duplicate rows are not eliminated. In either case, each row of the derived table is a row from either TABLE1 or TABLE2. By default, the Set Quantifier is

DISTINCTSyntax

```
<query expression body> UNION [ <set quantifier> ] <query term>
```

```
<query expression body> ::= <non-join query expression> | <joined table>
```

```
<query primary> ::= <non-join query primary> | <joined table>
```

```
<set quantifier> ::= DISTINCT | ALL
```

According to the SQL standards, each corresponding column of both queries must have the same column descriptor in order for two queries to be union-compatible.

Example

This operation results in the join with the shedding of the duplicates. The following query returns the phone numbers of schools and students from School and Student tables.

```
select School.PhoneNumber FROM School
```

UNION

```
select Student.PhoneNumber FROM Student
```

Phone No.
(219) 248 – 8261
(408) 615 – 7297
(408)615-2250
...
1-828-675-4262

In the above result, first three and the last record is displayed.

If the **ALL** is used with *UNION* operator, it retains the duplicate records. The syntax for the *UNION* with **ALL** though remains the same as shown in the following query.

```
select School.PhoneNumber FROM School
```

UNION ALL

```
select Student.PhoneNumber FROM Student
```

Result

Phone No.
1-828-675-4262
(408)615-2250
(408) 615 - 7297
...
(440) 238 - 7297

In this case, the result of the *UNION* queries, with the use of *ALL* or *DISTINCT* yields the same result, because there are no duplicate records.

INTERSECT OPERATOR

The **INTERSECT** works opposite to the **UNION** operator. Unlike the *UNION* operator which outputs the distinct records, the *INTERSECT* operator displays the records which are common for both the queries. When **ALL** is used with *INTERSECT* (that is, *INTERSECT ALL*), duplicate rows are not eliminated. In either case, each row of the derived table is a row from either *TABLE1* or *TABLE2*. By default, the *Set Quantifier* is **DISTINCT**

Syntax

```
<query expression body> INTERSECT [ <set quantifier> ] <query primary>
```

```
<query expression body> ::= <non-join query expression> | <joined table>
```

```
<query primary> ::= <non-join query primary> | <joined table>
```

```
<set quantifier> ::= DISTINCT | ALL
```

Example

```
select ClassID from Classes
```

INTERSECT

```
Select ClassID from Student
```

Result

ClassID
1
2

The above query displays the results in which the **ClassID** field have same values for both the tables. If **ALL** is used with **INTERSECT** operator, it retains the duplicate records. The syntax for the *INTERSECT* with **ALL** though remains the same as shown in the following query.

```
select SubjectID from ClassProperties
```

```
INTERSECT ALL
```

```
select SubjectID from Marksrecord
```

Result

SubjectID
1
1
1
1
...
6
6
6

Because of the **ALL** keyword, after performing the **INTERSECT**, the lesser number of copies of the records with same value, from either of the two tables, are displayed. In the above query , since, the number of records with value 2, in the *ClassProperties* is more than the number of records with the same value in the *Marksrecord*, in the final result, three records are displayed.

ORDER BY

The **ORDER BY** clause is an optional element of a **SELECT** statement which allows you to specify the order in which rows appear in the Result Set.

The rows are sorted first according to the first column specified in the *Order By* clause. If there are any duplicate values for this column, then the duplicate rows are sorted on the second column(within the first column sort) in the *Order By* list , and so on. **ASC** and **DESC** request the sorting in Ascending and Descending order respectively. By default, the values are sorted in Ascending Order.

Syntax

```
<order by clause> ::= ORDER BY <sort specification list>
```

```
<sort specification list> ::= <sort specification>
```

```
[ { <comma> <sort specification> }... ]
```

```
<sort specification> ::= < expression > [ <ordering specification> ]
```

```
<ordering specification> ::= ASC | DESC
```

Sort specification list:

Sort Specification list consist of the column names by which the ordering needs to be done.

Example

```
select StudentId, RollNumber from Student Order By RollNumber
```

Result

StudentID	RollNumber
1	1001
2	1002
3	1003
4	1004

The result shows first four records satisfying above query.

The above query, being the simplest one, lists the ID and Roll Numbers of the Students from the *STUDENT* table. The result will be sorted by Roll Number in the Ascending order, by default.

```
Select StudentId, StudentName,
```

```
RollNumber from Student
```

```
Where StudentName like 'Ca%'
```

```
Order By RollNumber DESC, StudentID
```

Result

StudentID	StudentName	RollNumber
3	Cathe	1003
1	Catherine	1001

This query lists the ID, Name and RollNumber of the students whose name starts with 'Ca'. The list is sorted by the *RollNumber* field in the Descending order and in case of duplicate records in the *RollNumber* field; the duplicate rows are sorted in the Ascending order according to *StudentId* field.

ALIAS SUPPORT

Alias Support is extended in the queries with the use of 'AS'. In a *select* expression, AS is used to assign an alias to the column name in the *select list* and table name in the *from* clause.

Syntax

<as clause> ::= [AS] <column name>

Example

```
select TeacherName, DateOfJoining, DateOfBirth as DOB from Teacher
```

Result

TeacherName	DateOfJoining	DOB
Mr. Agregado	1996-04-17	1965-04-10
Mr. Brumfield	1997-07-01	1966-11-27
Ms. McKelvey	1998-09-25	1968-01-07
Mr. Everett	1998-10-25	1968-01-17

The above query sorts the list of Teachers by their *DateOfBirth*, selecting the Name of the Teacher, his/her Date of joining and Date of Birth. The *DateOfBirth* is aliased as DOB which is then used in the Order By clause.

```
select  a.EmployeeID as EmployeeID, a.DepartName as Department, b.PostName
from    Teacher as a, Post as b
where a.PostID = b.PostID
```

Result

EmployeeID	Department	PostName
1	English	Principal
2	Science	Vice Principal
3	Science	Teacher
4	Math	Teacher
5	Computer	Teacher
6	English	Teacher
7	Social Studies	Teacher
8	Biology	Teacher

This query clearly shows the aliasing in the *select* statement as well as *from* clause.

Comments Support

Daffodil DB supports Comments in SQL queries.

Syntax

<comment> ::= <simple comment> | <bracketed comment>

<simple comment> ::= <simple comment introducer> [<comment character>...] <newline>

<simple comment introducer> ::= <minus sign><minus sign>[<minus sign>...]

<bracketed comment> ::= <bracketed comment introducer> <bracketed comment contents>
<bracketed comment terminator>

<bracketed comment introducer> ::= /*

<bracketed comment terminator> ::= */

<bracketed comment contents> ::=
[{ <comment character> | <separator> }...]

<comment character> ::= <nonquote character> | <quote>

<newline> ::= ;

Two types of comment are supported in Daffodil DB.

- 1) Simple Comment
- 2) Bracketed Comment

Simple Comment: A Simple Comment starts with 2 <minus sign> (--) and terminated with a semicolon (;). Examples of Simple Comments are

Select * From --This is a Select Query Selecting All Records of a Table Student; Student

Delete From --Name of table; Student where --Condition for Delete; Studentid<10

Bracketed Comment: A Bracketed Comment starts with a <solidus> (/) immediately followed by a <asterisk> (*) i.e. /* and terminated with <asterisk> immediately followed by a <solidus> i.e. */. Examples of Bracketed comments are

Select * From /* This is a Select Query Selecting

All Records of a Table Student */ Student

Update Student /* Set name of student to John where Studentid is 10 */ Set StudentName = 'John'
where Studentid =10

Call Statement

Call statement is used to invoke the SQL stored procedure.

Syntax

<call statement> ::= CALL <routine invocation>

<routine invocation> ::= <routine name> <SQL argument list>

<SQL argument list> ::=

<left paren> [<SQL argument> [{ <comma> <SQL argument> }...]] <right paren>

SQL Argument

SQL Argument is any valid SQL Expression.

Call Statement will automatically search for the matching SQL stored procedure. Matching stored procedure means the procedure that has the same name and matching <SQL argument list>.

Example

This is an example of CALL statement to execute the previously defined SQL stored procedures.

```
CALL Student_row_insert (11,'john',111,'m','xyz','24245',1)CALL Modify_Teacher_Salary  
(1,12000)CALL Student_Mark_InOut_Proc (5)
```

Session and Transaction Control Statements

Set Transaction Statement

This statement is used to define the Isolation Level.

Syntax

```
SET <transaction characteristics><transaction characteristics>::= TRANSACTION <transaction mode> [ { <comma> <transaction mode> }... ] }
```

Transaction Mode

Transaction mode can be Read Only and Read Write. In Read Only, we can only perform DQL (Data Query Language) Statements. In Read Write Mode, we can perform all the SQL Statements. A Transaction mode can also be used to set the various isolation levels like Read Committed, Read Uncommitted etc.

Example

Set Transaction Isolation Level READ COMMITTED

In the above example, we set the Transaction Isolation Level to READ COMMITTED.

Savepoint Statement

This statement is used to set a save point, or marker, within a transaction.

Syntax

```
SAVEPOINT <savepoint-name>
```

The Savepoint defines a location to which a transaction can return if part of the transaction is conditionally cancelled. If a transaction is rolled back to a savepoint, it must proceed to completion, or it must be cancelled altogether (by rolling the transaction back to its beginning). To cancel an entire transaction, use the form:

ROLLBACK

All the statements or procedures of the transaction are undone until last commit.

Commit Statement

The COMMIT statement successfully terminates a Daffodil DB transaction.

Syntax

The **COMMIT** statement is used to end the current transaction and make permanent all changes performed in the transaction. The **COMMIT** statement successfully finishes a transaction. This statement also erases all savepoints in the transaction.

You cannot roll back a transaction after a COMMIT statement is issued because the data modifications have been made a permanent part of the database.

Rollback Statement

Rollback statement will undo all the changes made since the last completed Transaction i.e. since the last COMMIT or ROLLBACK statement called.

Syntax

ROLLBACK [WORK] [TO SAVEPOINT <Savepoint-Name>]

Rollback erases all data modifications made since the start of the transaction or to a Save point. It also frees resources held by the transaction. ROLLBACK without a Savepoint_name rolls back to the beginning of the transaction. A transaction cannot be rolled back after a COMMIT statement is executed.

Example

ROLLBACK to SAVEPOINT save_point1

Above example rollbacks all changes or data modifications made since the start of the transaction to a save point name save_point1.

ROLLBACK WORK

Above example rollbacks all changes to the beginning of the transaction.

Set Session Authorization

This statement is used to change the current user in session to another user.

Syntax

<set session user identifier statement>::= SET SESSION AUTHORIZATION <value specification><value specification> ::= <literal> | <general value specification>

Literal

Literal can be only Character String Literal. No other types are allowed

General Value Specification

Value specification can only return a valid user-name or a valid roll-name, if user-name or roll-name does not exist in database then error is thrown. If we specify the values as current Date, current Database, current user and current time, then the function will throw an exception. Specify one or more values, host parameters, or SQL parameters.

Example

SET SESSION AUTHORIZATION administrator

Suppose a user 'daffodil' is connected to the database, after executing the above statement current user will change to 'administrator'.

Set Session Characteristics Statement

The SET SESSION CHARACTERISTICS statement is used to set one or more characteristics for the current SQL session. Characteristics include transaction modes read only, read write and isolation levels.

Syntax

```
<set session characteristics statement>::=SET SESSION CHARACTERISTICS AS <session  
characteristiclist><session characteristic list> ::=<session characteristic> [ { <comma> <session  
characteristic> }... ]<session characteristic> ::=<transaction characteristics><transaction  
characteristics> ::= TRANSACTION <transaction mode> [ { <comma> <transaction mode> }... ]
```

Session Characteristic List

Session Characteristics List is a collection of session characteristics separated by commas. Session Characteristics is in turn a transaction characteristic.

Transaction List

Transaction List is a collection of transaction modes separated by commas.

Transaction Mode

A Transaction mode can be Read Only and Read Write. In Read Only, we can only perform DQL Statements. In Read Write Mode, we can perform all the SQL Statements. A Transaction mode can also be used to set the various isolation levels.

Example

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL  
SERIALIZABLE, READ ONLY
```

After execution of this statement the isolation level of session is set to SERIALIZABLE and mode is set to READ ONLY.

SQL Security and Privileges

Schemas are used for controlling security in Daffodil DB. When creating a user, they do not have any access privileges to schemas of other users or other data objects within the database. The Daffodil DB RDBMS only permits the schema or database owner to grant privileges on the data objects within the schema.

Users can grant privileges to the following data objects in the schema:

- Tables
- Columns
- Roles
- SQL Procedures
- Domain

The following table describes the privileges that users can grant to other users for tables and columns.

DELETE	Allows a user to delete rows from tables within the schema
INSERT	Allows a user to insert rows of data into tables within the schema
REFERENCES	Allows a user to set up references to primary keys within the schema
SELECT	Allows a user to select rows from tables within the schema
TRIGGER	Allows a user to create triggers on tables within the schema
UPDATE	Allows a user to update rows in tables within the schema

Predefined User

Daffodil DB provides you with a predefined user (default user). User name and password of default user are –

User Name – PUBLIC

Password – PUBLIC

Default user i.e. “PUBLIC” can perform following operation with the database.

Connect to the database

Access the objects having rights to PUBLIC.

Granting and Revoking Privileges to Users

When you initially create a Daffodil DB database, it automatically creates a default user with a password “PUBLIC”. The user created owns the default schema USER. For security reasons, Daffodil DB does not recommend using this schema to store sensitive data.

Like any other user, default user must be granted the appropriate privileges to access data objects in schemas owned by other users. Current user will own any new schema that is created unless otherwise specified while creating the schema. New users are then able to create their own new schema and grant appropriate privileges on objects in the schema that they own. All new users must be granted privileges to access the objects in the USER schema, if this is required. To grant the ability for a user to pass a privilege on to other users, you must specify the optional WITH GRANT OPTION qualifier when granting the privilege.

Grant Statement

Use the GRANT statement to grant privileges on a data object.

Syntax

```
GRANT <privileges> TO <grantee> [ { <comma> <grantee> }... ] [ WITH GRANT OPTION ] [ GRANTED BY <grantor> ]
```

```
<Privilege> ::= <object privileges> ON <object name>
```

```
<object privileges> ::= ALL PRIVILEGES | <action> [ { <comma> <action> }... ]
```

```
<object name> ::= [ TABLE ] <table name> | DOMAIN <domain name> | <specific routine designator>
```

```
<action>:: =
```

```
SELECT
```

```
| SELECT <left paren> <privilege column list> <right paren>
```

```
| DELETE
```

```
| INSERT [ <left paren> <privilege column list> <right paren> ]
```

```
| UPDATE [ <left paren> <privilege column list> <right paren> ]
```

```
| REFERENCES [ <left paren> <privilege column list> <right paren> ]
```

```
| USAGE
```

```
| TRIGGER
```

```
| EXECUTE
```

```
<grantor> ::= CURRENT_USER | CURRENT_ROLE
```

```
<grantee> ::= PUBLIC | <authorization identifier>
```

- If you do not include one or more of these privileges in the GRANT statement, an error will be raised.
- If the optional “column-names” are not specified for the SELECT, INSERT, UPDATE, REFERENCES and TRIGGER privileges, the GRANT is a table-level grant that allows access to all present and future columns of the table.
- If you execute a GRANT statement that contains privileges that you don’t have or for which you do not have the right to grant, then an error occurs.
- You may only grant the EXECUTE privilege on an SQL Procedure.
- If you do not specify WITH GRANT OPTION, the user cannot pass the same privilege on to other users. However, if you do specify WITH GRANT OPTION, you have given the user permission to pass on the privilege to other users.
- Granting a privilege to PUBLIC grants the privilege to all present and future users. PUBLIC is a keyword, representing all users in the database.
- If you grant a privilege twice, and one of the times—either first or second—you granted the optional WITH GRANT OPTION and the other time you granted it without the grant option, the user will retain the grant option.
- If GRANTED BY <grantor> is not specified, then the grantor is the CURRENT_USER.

Example 1

The following statement grants SELECT privilege on the *TEACHER* table to the user USER1.

```
GRANT SELECT ON teacher TO user1
```

Example 2

Following GRANT statement allows the user ‘USER2’ to delete, insert and update rows from the *TEACHER* table; it also allows this user to grant same privileges to others.

```
GRANT DELETE,INSERT,UPDATE ON teacher TO user2 WITH GRANT OPTION
```

Example 3

Following GRANT statement allows the user ‘USER3’ to have ALL PRIVILEGES on the table *TEACHER*. However, the user ‘USER3’ will only be granted privileges that the user granting privileges has the rights to grant.

For example, if a user granting the privileges does not have right to grant DELETE privileges, the USER3 will not have the delete privilege.

```
GRANT ALL PRIVILEGES ON teacher TO user3
```

Example 4

Following GRANT statement allows the user ‘USER2’ to create trigger on *TEACHER* table; it also allows this user to grant same privileges to others.

```
GRANT TRIGGER ON teacher TO user2 WITH GRANT OPTION
```

Example 5

Following GRANT statement allows the user 'USER2' to use *TEACHER* table as referenced table; it also allows this user to grant same privileges to others.

GRANT REFERENCES ON teacher TO user2 WITH GRANT OPTION

Example 6

Following GRANT statement allows the users 'USER2' and 'USER3' to use *intdom* domain as a data type in any object where domain can be used.

GRANT USAGE ON intdom TO user2, user3

Example 7

Following GRANT statement allows the users 'USER1' and 'USER3' to execute procedure *Modify_Teacher_Salary* using call statement.

GRANT EXECUTE ON specific procedure Modify_Teacher_Salary TO user1, user3

Example 8

The following statement grants SELECT privileges on the 'employeeId' column and INSERT privileges on the 'postId' column of *TEACHER* table to the user USER3.

GRANT SELECT (EmployeeId), INSERT (PostId) ON teacher TO user3

Note: In above examples give different-different privileges on different-different objects to existing users. Roles can also be assigned privileges for different-different objects in the place of users.

Revoke Statement

Revoke Statement is used to revoke a role or a privilege from a user.

Syntax

REVOKE [GRANT OPTION FOR] <privileges> FROM

<grantee> [{ <comma> <grantee> }...]

[GRANTED BY <grantor>]

<drop behavior>

<drop behavior>: CASCADE | RESTRICT

To revoke a role from a user, use the SQL command, REVOKE. This command revokes *only* the privileges that the specified <grantor> granted to the <grantee>. If another <grantor> granted the same privileges to the <grantee>, then the <grantee> will still have those privileges.

Note: - The syntax rule for the REVOKE syntax is similar to the GRANT statement. The major difference is the additional RESTRICT or CASCADE keyword and the GRANT OPTION FOR clause. The following describes the optional clauses GRANT OPTION FOR and RESTRICT or CASCADE.

You may only revoke privileges, which you have granted.

GRANT OPTION FOR

If the optional GRANT OPTION FOR clause is used, the WITH GRANT OPTION right is revoked, but the actual privilege itself is not revoked then CASCADE and RESTRICT may be used in the same way as the normal REVOKE statement.

RESTRICT | CASCADE

If you specify the RESTRICT keyword, only privilege granted by you, will be revoked from the specified user. If the specified user had grant option and has granted the same privilege to other users, then there will be an error. If you specify CASCADE, only the privileges granted by you, will be revoked from the specified user or any other privileges dependent on your grant.

Example 1

Following statement revokes the SELECT privilege on the *TEACHER* table from the user USER1.

```
REVOKE SELECT ON teacher FROM user1 restrict
```

Example 2

The following REVOKE statement removes ALL PRIVILEGES from the user, USER3 on the table *TEACHER*.

```
REVOKE ALL PRIVILEGES ON teacher from user3 restrict
```

Example 3

The following REVOKE statement revokes the select privileges on column 'EmployeeId' and insert privileges on column 'PostId' on the *TEACHER* table from the user 'User3'.

```
REVOKE SELECT (EmployeeId), INSERT (PostId) ON teacher from user3 restrict
```

Example 4

The following REVOKE statement revokes 'grant option for' ALL PRIVILEGES on the table *TEACHER* from the user 'User3'. After it user 'User3' could not grant any privileges on table *TEACHER* to any existing user/role.

```
REVOKE GRANT OPTION FOR ALL PRIVILEGES ON teacher from user3 cascade
```

CREATE ROLE

Creates a role to which the privileges can be granted.

Syntax

```
CREATE ROLE <role_name> [WITH ADMIN <grantor>]  
    <grantor> ::= CURRENT_USER | CURRENT_ROLE
```

role_name

It is the name of the role you are creating. For <role_name>, you can not use any existing user name and reserve words.

WITH ADMIN <grantor>

- If WITH ADMIN <grantor> is not specified, then the grantor is the CURRENT_USER.
- IF WITH ADMIN CURRENT_ROLE is specified, then the CURRENT_ROLE must not be NULL.

Examples**CREATE ROLE PRINCIPAL WITH ADMIN CURRENT_USER**

If current user is USER1, this will create a role called PRINCIPAL whose owner will be the user USER1. Privileges can now be granted to the role PRINCIPAL. The user USER1 can then grant this role PRINCIPAL to other users, or to other roles. Once the role is granted to the users or to other roles, these users and roles will have same level of privileges as was granted to the role PRINCIPAL.

GRANT ROLE

Use GRANT ROLE statement to grant role to users or to other roles.

Syntax

```
GRANT <role_name> [{, <role_name>} ...]  
TO <grantee> [{, <grantee>} ... ]  
[WITH ADMIN OPTION]  
[GRANTED BY <grantor>]  
<grantee> ::= PUBLIC | <authorization identifier>  
<grantor> = CURRENT_USER | CURRENT_ROLE
```

role_name

It is the name of the role to be granted. You may grant more than one role.

Grantee

- A role can be granted to users or to other roles.
- You cannot grant a role to itself.
- You cannot grant one role to a second role, and then attempt to grant the second role back to the first. For example, you can grant Role (A) to Role (B) or Role (B) to Role (A), but not both. Such a series of grants would result in a role grant cycle, which is not allowed.
- Granting to PUBLIC grants the role to all present and future users and roles.

WITH ADMIN OPTION

- If WITH ADMIN OPTION is specified, then the <grantee> can grant the role to other users or roles.
- If you do not specify GRANTED BY <grantor>, then the grantor is the CURRENT_USER.
- If you specify GRANTED BY CURRENT_ROLE, then the current role must not be NULL.

Note: - To successfully execute this command, current users or roles must either be the role owner. Or, the <grantor>s must have admin option for every role that they grant.

Examples**GRANT PRINCIPAL TO USER2
WITH ADMIN OPTION GRANTED BY CURRENT_USER**

If current user is USER1:-

This will grant a role called PRINCIPAL (whose owner is the user USER1) to another user USER2 with admin option. The user USER2 can grant the role PRINCIPAL to other users, or to other roles, because the user USER2 has 'with admin option' for role PRINCIPAL.

REVOKE

Use REVOKE to revoke a role from a user or another role. This command revokes only the roles that the specified <grantor> granted to the <grantee>.

Syntax

```
REVOKE [ADMIN OPTION FOR] <role_name> [{, <role_name>} ...]  
FROM <grantee> [{, <grantee>} ...]  
[GRANTED BY <grantor>]  
<drop behavior>  
<drop behavior> ::= CASCADE | RESTRICT
```

Please note that the syntax rule for the REVOKE syntax is similar to GRANT ROLE, except for the following.

NOTE: You may only revoke roles, which you have granted.

ADMIN OPTION FOR

If ADMIN OPTION FOR is specified, then only the admin option for the role is revoked.

Drop behavior

- If you specify the RESTRICT keyword, If the specified <grantee> had the ADMIN OPTION and granted the same privilege to other users, then privileges will be retained otherwise revoked.
- If you specify CASCADE, only the role granted by you, will be revoked from the specified <grantee> and any other roles dependent on your grant.

Examples

REVOKE PRINCIPAL from USER3 restrict

If the current user is Marty:

This will revoke a role called PRINCIPAL (whose owner is the user Marty) from USER3.

REVOKE ADMIN OPTION FOR PRINCIPAL from USER2 granted by CURRENT_USER cascade

If the current user is USER1:

This will revoke a role called PRINCIPAL (whose owner is the user USER1) from USER2.

DROP ROLE

Used to drop an existing role. To successfully execute this command, the current user must be a user who is the owner of the role.

Syntax

DROP ROLE <role_name>

Examples

DROP ROLE URole

If the current user is USER1:

This will drop the role called URole whose owner is the USER1 (Role owner).

NOTE: You may only drop roles, which you have created.

SET ROLE**Syntax**

SET ROLE <role_name>
| NONE

Usage Notes

- To successfully execute this command, the current user must be the role owner, or a user granted to use this role.
- This statement will set the current role for the current user to either the role specified or to the null value if NONE is specified.

Example 1

The following statement will set the role 'Principal' and after that only those schema objects could be accessed, for which the roles have been set for.

SET ROLE PRINCIPAL

Appendix**1. Error Messages**

DSE0={0}

DSE12=Access denied. Do not have 'CREATE or DROP' permission.

DSE14=An aggregate may not appear in the where clause unless it is in a subquery contained in a having clause or a SELECT list, and the being aggregated is an OUTER reference.

DSE15=OLD and NEW alias name cannot be identical.

DSE16=Function {0} not supported.

DSE17=Ambiguous column name {0}.

DSE22=Feature {0} not supported.

DSE27=Insufficient privileges.

DSE28=In auto commit mode.

DSE35=Syntax error converting data type {0} to {1}.

DSE40=Cannot concatenate character data type value {0} to byte data type value {1}.

DSE85=Cannot call Statement.executeUpdate() with this query.

DSE86=Cannot call Statement.executeQuery() with this query.

DSE87=Syntax error converting data type {0} to {1}.

DSE103=Cannot concatenate date with date.

DSE104=View or function {0} is not updatable because it contains aggregates.

DSE105=Cannot execute query through execute query method.

DSE146=Cannot read from the input stream.

DSE191=Cannot be dropped as this procedure is being referred by some other procedure.

DSE198=Cannot define the SQL data access as no-SQL.

DSE200=A routine definition can have at most one <deterministic characteristic>.

DSE201=A routine definition can have at most one <SQL-data access indication>.

DSE202=A routine definition can have at most one <dynamic result sets characteristic>.

DSE203=A routine definition can have at most one <specific name>.

DSE205=A routine definition can have at most one <language clause>.

DSE206=A routine definition can have at most one <parameter style clause>.

DSE210=Cannot specify <parameter mode> in function definition.

DSE212=Cannot use name and index in same statement.

DSE213=Cardinality cannot be greater than one.

DSE214=Cardinality does not match.

DSE224=Catalog name in constraint definition does not match catalog of table definition.

DSE225=Catalog name can not be changed as table name or schema name does not exist.

DSE226=Catalog name in schema definition and table definition for table {0} does not match.

DSE227=Catalog name in the view definition is not matching with that of schema definition.

DSE228=Catalog name of the trigger definition is not matching with that of schema definition.

DSE231=Character string type can not be null.

DSE232=Characteristics cannot be null.

DSE235=Check constraint definition does not exist.

DSE236=Check option cannot be specified with recursive view type.

DSE237=Check the code.

DSE239=Check datatype called from Boolean value expression and Boolean value expression.

DSE242=Callable statements are not supported.

DSE243=Collate clause can not specified for non-character column type.

DSE250=Column names in each table must be unique. Column name {0} in table {1} is specified more than once.

DSE251=ALTER TABLE DROP COLUMN statement failed because column {0} does not exist in table {1}.

DSE252=ALTER TABLE DROP COLUMN statement failed because {0} is the only data column in table {1}. A table must have at least one data column.

DSE255=Column name {0} does not exist in target table {1}.

DSE256=Invalid column name(s) {0}.

DSE260=Invalid column name(s) {0} in Group by clause.

DSE262=Column type not set.

DSE264=Column descriptor does not exist. TABLE CATALOG {0} TABLE SCHEMA {1} TABLE NAME {2} COLUMN NAME {3} PRIVILEGE TYPE {4}.

DSE265=Table descriptor does not exist. TABLE CATALOG {0} TABLE SCHEMA {1} TABLE NAME {2} PRIVILEGE TYPE {3}.

DSE266=Column {0} does not exist in table {1}.

DSE267=Column name {0} passed with value {1}.

DSE269=Column name specified in columnlist is not present.

DSE270=Columns corresponding to the constraint not found {0}.

DSE272=Cannot use duplicate column names in index key list. Column name {0} listed more than once.

DSE273=Columns specified in trigger column list are not unique.

DSE274=Trigger {0} uses the invalid column/columns {1} in trigger column list.

DSE275=Commit action can not be specified with type {0}.

DSE276=Commit is not allowed with read uncommitted level.

DSE279=Connection already closed.

DSE286={0} is not a constraint.

DSE287=Non deferrable constraint.

DSE288=ALTER TABLE DROP CONSTRAINT statement failed because constraint {0} is being referenced foreign key constraint {1}.

DSE289=Unique constraint referred by the referencing column should not be deferrable.

DSE294=Current row is not valid.

DSE295=Data cannot be loaded.

DSE297=Invalid data.

DSE301=Data not loaded in the {0} descriptor.

DSE306=Data not valid.

DSE310=Data type descriptor does not have any rows.

DSE316=Database {0} does not exist.

DSE314=Database {0} already exists.

DSE319=Data type should be of character type in case collate clause is specified.

DSE323=Default value of the domain descriptor is not set.

DSE324=Definition of rule {0} is not found.

DSE325=Degree can not be greater than one.

DSE326=Degree is exceeding to one in between predicate.

DSE330=Deletion is not allowed with read uncommitted level.

DSE334=Divide by zero error encountered.

DSE336=Domain constraint descriptor not loaded.

DSE337=Domain definition does not exist.

DSE338=Domain descriptor not set.

DSE339=Domainconstraint does not exist. Catalog: {0} schema: {1} constraint: {2}.

DSE340=Do not have permission to {0} the table {1}.

DSE342=Do not have any permission to SELECT a row.

DSE355=Error while getting the connection.

DSE356=Error in reading the stream.

DSE358=Exception.

DSE382=INSERT statement conflicted with FOREIGN KEY constraint {0}. The conflict occurred in table {1}.

DSE390=Global session not set.

DSE393=exception not handled correctly.

out-of-range datetime value.

DSE396={0} specified should be between {1} and {2}.

DSE401=Illegal call.

DSE402=Illegal call for condition: {0}.

DSE408=Illegal condition for join retriever optimal {0}.

DSE410=Database is incompatible, might be created in higher version.

DSE411=Illegal method call.

DSE412=Illegal multiplier passed {0}.

DSE413=Illegal timestamp denominator {0}.

DSE415=Invalid fetch direction. In case of type forward, only fetch forward is valid direction.

DSE416=Incompatible type {0}.

DSE419=Illegal/invalid argument passed in function {0}.

DSE478=Index cannot be maintained on column of type {0}.

DSE479=Index passed [{0}] must be >0 && <= {1}.

DSE482=Cannot find Index name {0}.

DSE483=Cannot defer a constraint that is not deferrable.

DSE484=Insert failed.

DSE487=Insertion is not allowed with read uncommitted level.

DSE489=Integrity constraint violation.

DSE492=Internal error data type descriptor is not set or is null.

DSE493=Internal error: columndescriptor not set.

DSE494=Internal exception: table descriptor not initialized.

DSE495=Internal exception: schema descriptor not set.

DSE496=Internal exception: column descriptor not set.

DSE504=Invalid column.

DSE505=Invalid column index: {0}.

DSE508=Invalid column name {0}.

DSE511=Invalid concurrency {0}.

DSE513=Invalid data passed.

DSE514=Invalid data type {0}.

DSE515=Cannot find data type/domain {0}.

DSE517=Invalid default option <{0}> {1}.

DSE518=Invalid fetch size.

DSE520=Invalid grantor.

DSE523=Invalid log file.

DSE524=Invalid logging level.

DSE525=Invalid method call.

DSE526=Invalid no of columns in function.

DSE527=Invalid number {0}.

DSE528=Invalid operator {0}.

DSE529=Invalid operator (-) for string manipulation.

DSE530=Invalid quantifier {0}.

DSE531=Invalid query.

DSE532=Invalid query to execute.

DSE533=Invalid query to execute update method.

DSE536=Invalid result from server.

DSE537=Syntax error converting datetime from character string.

DSE538=Invalid URL.

DSE540=This feature is not supported in ONE\$DB.

DSE541=Invalid value for fetch direction: {0}.

DSE542=Invalid value for holdability {0}.

DSE543=Invalid value for isolation level {0}.

DSE545=Invalid value passed for maxrows {0}.

DSE548=Iterator is not at valid row.

DSE550=Key can not be null.

DSE552=Key does not exist {0}.

DSE554=Key passed is null.

DSE555=Keys cannot be null key1 {0} key2 {1}.

DSE558=Length cannot be null.

DSE562=List is empty.

DSE563=Locator not allowed in procedures.

DSE565={0} not supported.

DSE567=Method can not be called with parameters.

DSE715=Neither schema name nor authorization identifier specified.

DSE716=NEW ROW alias cannot be specified with DELETE type of event.

DSE717=NEW ROW cannot be specified without FOR EACH ROW.

DSE718=NEW TABLE alias cannot be specified with DELETE type of event.

DSE720=Next of {0} is null.

DSE721=Column name not set

DSE723=No data exists corresponding to the data.

DSE725=No getter method called before this call.

DSE728=Cannot find the procedure {0} which has {1} arguments.

DSE729=Cannot drop the procedure {0}, because it does not exist.

DSE731=The variable name {0} has already been declared. Variable names must be unique within a stored procedure.

DSE732=No type available.

DSE736=Not a select query.

DSE737=Not a valid column.

DSE738=Not a valid direction {0}.

DSE740=Not a valid out parameter index {0} outparameters {1}.

DSE749=ColumnNames are null.

DSE750=Null not allowed in this column.

DSE752=Number of columns in query expression and view column list are not equal.

DSE753=Procedure parameters mismatch.

DSE754=Number of values are not equal to number of columns

DSE756=Object not convertible

DSE759=Object privilege should be of usage type in this case

DSE773=The value you entered is not consistent with the data type or length of the column.

DSE775=Old or old row, new or new row, old table, and new table shall be specified at most once each within the <old or new values alias list>.

DSE776=Old row alias can not be specified with insert type of event.

DSE777=Old row can not specified without for each row.

DSE778=Old table alias can not be specified with insert type of event.

DSE784=Parameter passed is {0}.

DSE785=Parameters and values count does not match.

DSE786=Parameters are null.

DSE792=Parameterstyleclause cannot be specified in procedure statement.

DSE798=Position must be greater then 0.

DSE799=Precision {0} can not be greater maximum value {1}.

DSE804=Problem while checking for index table.

DSE806=Problem while creating the database.

DSE808=Problem in getting retriever for table.

DSE810=Problem in getting table {0}.

DSE812=Problem in getting the constraints of the table.

DSE813=Problem in run method of default option.

DSE818=Put quotes arround the values passed.

DSE819=Query not produced an update count.

DSE820=Query not produced ResultSet.

DSE822=Query expression not declared local temporary table.

DSE823=Query expression can not have target specifications.

DSE824=Query expression shall specify row type of element.

DSE825=Query failed.

DSE828=Query is either null or empty.

DSE832=Query time out must be greater than 0. value passed is {0}.

DSE836=Range is exceeding {0} start position {1} is greater than length of blob {2}.

DSE841=Record is not present.

DSE848=Record number is exceeding the total number of records {0}.

DSE849=Record number {0} passed is deleted.

DSE851=Recursive shall be specified.

DSE853=Referenceable view specification can not be specified with recursive.

DSE857=Referential constraint definition does not exist.

DSE861=Restrict called from {0}.

DSE869=Result can be specified at most one time.

DSE870=ResultSet not updateable.

DSE871=ResultSet type is forward only.

DSE874=Invalid role specification.

DSE875=Role authorization descriptor does not exist. Role name: {0} grantee: {1}.

DSE876=Role/user {0} already exists in database.

DSE877=Rollback is not possible, not a valid session.

DSE879=Row is locked by another user.

DSE883=Rowset is read-only.

DSE884=Rsb of rsbi {0}, record.

DSE885=Rule name passed is {0}.

DSE887=Runtime exception: table descriptor not set.

DSE889=Save point {0} already exists.

DSE890=Invalid save point.

DSE891=Scale {0} can not be greater maximum value {1}.

DSE892=Scale {0} can not be greater precision {1}.

DSE893={0} scale can not be negative.

DSE895=Schema and catalog specified does not exist.

DSE896=Schema {0} does not exist.

DSE897=Schema in the constraint definition does not match schema of table.

DSE902=Schema name does not match with that of table {0}.

DSE903=Schema name does not match with that of view definition.

DSE904=Schema name not specified.

DSE905=Schema name does not match with that of trigger definition.

DSE906=Schema name's domain definition does not match.

DSE907=Schema owned by {0} has not granted drop privilege to this user.

DSE915=Session does not exist.

DSE918={0} should contained either in aggregate function or group by columns.

DSE920=Should specify parameter name.

DSE925=Some problem.

DSE928=Specific name is already present in the schema.

DSE930=Specified table is not a base table {0}.

DSE931=Splitindexandnonindexconditions called from betweenpredicate.

DSE934=SQL invoked routines can only have execute privilege.

DSE937=Start position {0} is greater than the length of blobclob {1}.

DSE939=Start position {0} is greater than the length of blob {1}.

DSE940=Statement is closed.

DSE945=Sum or avg aggregate function can not accept character string as argument.

DSE946=Syntax error.

DSE953=Systemfield {0} does not exist.

DSE954=Ambiguous table name {0}.

DSE955=Table cannot be a local temporary one.

DSE956=Table definition does not contain column definitions.

DSE957=Table definition does not contain table elements.

DSE958=Table definition does not contain table content source.

DSE959=Invalid object name {0}.

DSE962=Table is not a base table. Alter table not allowed.

DSE963=Table is of reference able type.

DSE964=Table is referenced from the check constraints, can not be dropped.

DSE970=Table name does not exist

DSE972=Temporary table should have all privileges.

DSE973=The data type should be of Character String type but is of {0}.

DSE974=The data type does not require scale.

DSE975=Length {0} of column {1} can not be less than 128 for default clause

DSE976=Cannot find the index name {0} on table {1}.

DSE977=The is no corresponding index columns.

DSE978=The key is not present.

DSE982=The key passed is null or invalid.

DSE983=The length {0} can not be greater than valid length {1}.

DSE985=The length {0} is greater than valid length {1}.

DSE986=The list is empty.

DSE987=The literal length {0} is greater than {1}.

DSE989=The precision {0} is greater than valid precision {1}.

DSE991=The recordId {0} not an instance of query record.

DSE992=The record identity {0} not an instance of query record.

DSE994=The time precision {0} is greater than valid {1}.

DSE998=View {0} does not exist.

DSE999=There is no record corresponding to recordId {0}.

DSE1010=Transformgroupspecification cannot be specified in procedure statement.

DSE1012=You can specify trigger privilege only on base tables.

DSE1013=Trigger definition does not exist.

DSE1014=Trigger subject table is not a base table.

DSE1017=Truncated length {0} is greater than length of blobclob {1}.

DSE1018=Type {0}.

DSE1019=Inappropriate type {0}.

DSE1021=Type {0} not defined in database.

DSE1022=Type not registered.

DSE1024=mismatched type.

DSE1025=Types of table does not match.

DSE1026=Column {0} is not the same data type as referencing column {1}.

DSE1029=Unable to get the authorization identifier.

DSE1030=Unable to get the schema name.

DSE1032=Violation of UNIQUE KEY constraint {0}. Cannot insert duplicate key in object {1}.

DSE1036=Unsupported format.

DSE1037=Update cascade failed.

DSE1043=Update primary key returned by retriever cannot be null.

DSE1046=Usageprivilegedescriptor does not exist. Grantor: {0} grantee {1} OBJECT_CATALOG: {2} OBJECT_SCHEMA: {3} OBJECT_NAME {4} OBJECT_TYPE {5}.

DSE1047=User cannot be blank or _SYSTEM.

DSE1051=Value cannot be null.

DSE1066=View column list must be specified with recursive view type.

DSE1067=View definition does not exist.

DSE1075=Year must be between 9999 and -4713.

DSE1077=You cannot repeat a routine characteristic.

DSE1081=Illegal argument exception.

DSE1082=The column specified in view query is not valid.

DSE1083=Column {0} not allowed as constraint column. Type is {1}.

DSE1084=Cannot use connect in routine definition.

DSE1086=There are no primary or candidate keys in the referenced table {0} that match the referencing column list in the foreign key {1}.

DSE1087=The length of check clause {0} is greater than the permitted length {1}.

DSE1088={0} is System Field; change the name of the column.

DSE1090=Create View failed because no column name was specified for functional column.

DSE1103=Length of escape character can not be greater than one.

DSE1104=Recordsetbuffer {0} record buffer {1}.

DSE1105={0} is not convertible in binary.

DSE1107=Problem in getting view characteristic.

DSE1108=Problem in getting column characteristidefc of table {0}.

DSE1109=Problem in getting default clause for table {0}.

DSE1110=Problem in getting unique constraint.

DSE1111=Problem in getting check constraints.

DSE1112=Problem in getting trigger characteristics.

DSE1122=Problem in checking for has deferred constraints.

DSE1124=Update failed due to dataexception.

DSE1126=Delete failed due to retrievalexception.

DSE1128=Commit failed due to data exception.

DSE1129=Perform failed due to data exception.

DSE1130=Column {0} not found in characteristics of table {1}.

DSE1131=Column index {0} not found in characteristics of table {1}.

DSE1133={0} data type not supported.

DSE1134=Dependent privilege descriptor still exist.

DSE1135=There is already an object named {0} in the database.

DSE1136=Column {0} already exists in the table {1}.

DSE1137=There is already an object named {0} in the database.

DSE1138={0} type table privilege already exists for grantor {1} grantee {2} table {3}.

DSE1139={0} type column privilege already exists for grantor {1} grantee {2} table {3} column name {4}.

DSE1141=Trigger with {0} name already exists.

DSE1142=There is already an index on table {1} named {0}.

DSE1143=Column {0} already exists for index {1} of table {2}.

DSE1144={0} domain already exists.

DSE1145=Domain constraint {0} already exists.

DSE1146=Schema {0} already exists.

DSE1147=Column {0} already exists for constraint {1}.

DSE1148={0} type usage privilege already exists for GRANTOR {1} GRANTEE {2} object {3}.

DSE1149={0} type routine privilege already exists for grantor {1} grantee {2} routine {3}.

DSE1151=Grantee {0} already exists for role {1}.

DSE1152=Method specification already exists for SPECIFIC CATALOG {0} SPECIFIC SCHEMA {1} SPECIFIC NAME {2}.

DSE1153=Number of values is not equal to number of parameters.

DSE1157=Parameters not set properly or not required.

DSE1160=Argument passing is not proper.

DSE1164=Updation is not allowed with read uncommitted level.

DSE1166=Passed object length {0} is more than {1}.

DSE1167=Required object of {0} passed {1}.

DSE1168=Property {0} of column index {1} contains invalid value {2}.

DSE1171=No such privilege descriptor found to delete.

DSE1172=Invalid privilege descriptor.

DSE1173=Some dependent entry exist.

DSE1174=Incorrect syntax at position {0} near {1}.

DSE1175=Problem in adding column {0}.

DSE1178=Time out {0}.

DSE1179=Catalog name is not same in object name {0} and schema descriptor {1}.

DSE1180=Schema and table descriptor both are set.

DSE1181=Granted or revoked privilege {0} is not compatible with object.

DSE1182=No action found in action list.

DSE1183=Table descriptor is null.

DSE1184=Access mode is read only.

DSE1199=Constraint cannot be created as data exists in table.

DSE1205=Cannot add multiple PRIMARY KEY constraints to table {0}.

DSE1206=File growth can be from 10 - 100 only is ?.

DSE1207=Insufficient privileges to drop the database.

DSE1208=Invalid username/password.

DSE1209=Insufficient privileges.

DSE1210=User {0} with password {1} does not exist.

DSE1211=Table is locked by another user.

DSE1214=Key is not valid.

DSE1216=Invalid status{0}.

DSE1219=Cannot insert in functional/aggregate columns.

DSE1249=Not a select query.

DSE1250=Cardinality for multiple rows is invalid.

DSE1251={0} statement conflicted with CHECK constraint {1} defined as {2}. the conflict occurred in table {3}.

DSE1252=Length of columns {0} not equal to length of REFERENCES {1}.

DSE1254=Problem in firing replace event.

DSE1255=Violation of PRIMARY KEY constraint {0}. Cannot insert duplicate key in object {1}.

DSE1256=Insert error: column name or number of supplied VALUES does not match table definition.

DSE1257=The name {0} is not permitted in this context. Only constants, expressions, or variables allowed here. Column names are not permitted.

DSE1258=Type of join applied in the query is not valid.

DSE1259=Some column has been updated with value not satisfying the query.

DSE1260=Cursor [{0}] already created in the current SQL SESSION.

DSE1261=Cursor {0} already opened

DSE1262=Column cannot be of reference type.

DSE1263=Cannot UPDATE the CURSOR, CURSOR is not updatable.

DSE1267=Cannot perform INSERT/UPDATE/DELETE directly in a VIEW.

DSE1269=Mismatch in length of columns and values.

DSE1270=The query is not valid.

DSE1271=The query does not support ORDER BY.

DSE1272=Non-updatable type of column in where clause,cannot INSERT/UPDATE in this query.

DSE1273=Value of some reference column not provided or invalid column in query.

DSE1274=Problem in Execution of Trigger For Table {0} While Performing Action [{1}] due to {2}.

DSE1275=Problem in Execution of Insert Statement for Table {0}.

DSE1276=Problem in Execution of Update Statement for Table {0}.

DSE1277=Problem in Execution of Delete Statement for Table {0}.

DSE1278=Referenced Constraint Violation [RESTRICT] for Table = {0} .

DSE1284=INSERT statement conflicted with FOREIGN KEY constraint {0} MATCH PARTIAL.
Cannot update the NULL values in all referencing column(s).

DSE1285=INSERT statement conflicted with FOREIGN KEY constraint {0} MATCH PARTIAL.
Cannot insert the NULL values in all referencing column(s).

DSE1286=DELETE statement conflicted with FOREIGN KEY constraint {0}. The conflict occurred
in table {1}.

DSE1287=Referenced constraint violation.

DSE1288=UPDATE statement conflicted with UNIQUE KEY constraint {0}. The conflict occurred in
table {1}.

DSE1289=UPDATE statement conflicted with PRIMARY KEY constraint {0}. The conflict occurred
in table {1}.

DSE1290=UPDATE statement conflicted with FOREIGN KEY constraint {0}. The conflict occurred
in table {1}.

DSE1291=Cannot insert the value NULL into column {0}, table {1}; column does not allow nulls.
INSERT failed.

DSE1292=Cannot update the value NULL into column {0}, table {1}; column does not allow nulls.
UPDATE failed.

DSE1293=Table name does not exist in mapping.

DSE1294=TableDetails for table name {0} passed in event is not found.

DSE1295=There are {0} columns in the INSERT statement than values specified in the values clause. the number of values in the VALUES clause must match the number of columns specified in the INSERT statement.

DSE1300=Aggregate function cannot be used in the insert value.

DSE1301=Select statement can yield only one row.

DSE1303=Invalid column {0} has been used in trigger action for table {2}.

DSE1304=Invalid column for trigger specified in the trigger statement.

DSE1305=Primary or Unique column should be autoincremental.

DSE1306=Cannot insert a record with HASRecord columnvalue equal to false.

DSE1307=Target Specification cannot be null

DSE1308=Invalid user {0}

DSE1309=Invalid password

DSE2001=Now we have to insert record.

DSE2002=Key is not valid.

DSE2003=Record number {0} passed is deleted.

DSE2004=Record is not present.

DSE2005=Get the record from this location {0}.

DSE2006=Record is partial.

DSE2007=Record number is exceeding the total number of records {0}.

DSE2008=Key {0} value {1} pair not present.

DSE2009=Duplicate keys are not allowed.

DSE2010=Cluster is locked by another user.

DSE2012=Database is locked by another user.

DSE2013=Type {0}.

DSE2014=Column does not exist {0}.

DSE2015=There is no default index on table {0}.

DSE2016=Error while updating the index in table {0}.

DSE2017=Index {0} already exists for table {1}.

DSE2018=Number of values are not equal to number of columns.

DSE2019=Iterator is at invalid position, 1 : after last , -1 : before first.

DSE2023=Server is already closed {0}

DSE2025={0}.

DSE2027=Cannot add database with more than one file and numtfile support as FALSE.

DSE2028=Length of new files :: {0} initial size :: {1} increment factor is not possible with the database files u r trying to ADD.

DSE2029=Initial size must be greater than size of file exist.

DSE2030=File size :: {0}, initial size :: {1}, increment factor :: {2} not possible.

DSE2031=Page Size is invalid. Valid range is 4k-32k.

DSE2032=Column not found : {0}.

DSE2034=Table is in use : {0}.

DSE2035=No index created in table : {0}.

DSE2036=Index already exist.

DSE2037=Table name found.

DSE2038=Element is deleted.

DSE2039=Node size should be greater than 2.

DSE2040=Index table not initialized.

DSE2041=NODE IS NONLEAF

DSE2042=NO Valid Key In Node

DSE2043=POSITION PASSED IS GREATER THAN ELEMENT COUNT == POSITION {0} AND ELEMENT COUNT {1}

DSE2044=SPLITTING WILL OCCUR NOW

DSE2045=BTree Can't Give Value Of This Column

DSE2046=Database version is not Compatible

DSE2047=Increament Factor can't be negative

DSE2048=Encryption key length can't be greater than 256

DSE2049=DatabaseName {0} contains illegal character.

DSE2050=Dead Lock Detected.

DSE2051=Either path specified for database or database name is too long which exceeds OS limits.

DSE2052=Transaction is active - Unable to set isolation level - First commit or rollback.

DSE2053=Cursor can not be declared without a procedure.

DSE2054={0} is not supported in one dollar db.

DSE3513=Type of iterator is not Updatable

DSE3514=Invalid Column {0} for table {1}

DSE3515=Column {0} Not Found in Mapping

DSE3516=Table {0} Does Not Exist In Mapping

DSE3517=Invalid Column {0}

DSE3518=Position of Iterator Not Initialised

DSE3519=Illegal Column Type

DSE3520=Value Of COLUMN {0} Not Found in All Iterators

DSE3521=View Tables from SQL Hierarchy does not match fully with the Plan Hierarchy

DSE3522=Type Of Iterator {0} is not initialized

DSE3523=Illegal Call to Iterator

DSE3524=Clone not supported.

DSE3526=Table {0} does not lie in plan hierarchy.

DSE3527=Cost not caculated for Condition {0}.

DSE3528= ? is not allowed in Order BY and Group By Clause

DSE3529=Either of joinlevelorder {0} and sIngletablelevelorder {1} should present.

DSE3530=Condition {0} not solvable on any plan.

DSE3531=Cost not calculated for this plan.

DSE3532=Listener is not supported in case of set operator and subquery.

DSE3533=Columnnames specified in from sub query and derived column list are not equal in length.

DSE3534=Alias name should be specified with aggregate/expressional or scalar functional columns.

DSE3535=Duplicate columns specified in from subQuery or view.

DSE3536=Aggregate columns cannot be present in onCondition.

DSE3537=Relation {0} does not belong to any plans.

DSE3538=Condition {0} can not be shifted to single table level.

DSE3539=Wrong event type {0} passed.

DSE3540=HasRecord column cannot be used in select list when group by is present.

DSE3541=A column has been specified more than once in the order by list. Columns in the order by list must be unique.

DSE3542=Ambiguous column naming in select list.

DSE3543=ORDER BY items must appear in the select list if the statement contains a UNION operator.

DSE3544=Order Column {0} not present in selectList of query involving set operator.

DSE3545=The ORDER BY position number {0} is out of range of the number of items in the select list.

DSE3546=HasRecord Column contains invalid table name {0}.

DSE3547=Condition {0} Execution Plan is not initialised.

DSE3548=Columns and their order should be equal in length.

DSE3549=Data type and size is not initialized for scalar function.

DSE3550=? is not allowed in expression in selectlist

DSE3551=Aggregate COLUMNS cannot be present for insertion in insert statement.

DSE3552=Invalid Count value inside Top function.

DSE3553=Iterator is at Invalid status.

DSE3554=Child length can not be more than {0}.

DSE3555=Reference {0} Value is not found.

DSE3556=Execution Plan not initialized for table {0}.

DSE3557=Column {0} does not exist in column characteristics of table {1}.

DSE3558=Columns present in Comparison Predicate is Not Equal to 2.

DSE3559=Columns present in Comparison Predicate is Null.

DSE3560=Type of Aggregate function is not initialised.

DSE3561=Aggregate Function Quantifier type is not initialised.

DSE3562=References and Values passed are not equal in length.

DSE3563=Reference {0} not found in {1}.

DSE3564=Mapping is not properly initialized.

DSE3565=Inappropriate type in {0}

DSE3566=Having clause can not be given without any aggregate columns or group by

DSE3570=Invalid data type {0}.

DSE3571=Collator of column {0} and column {1} does not match, hence we cannot compare

DSE3572=No Proper Comparator...data type 1 {0} data type 2 {1}

DSE3573=No value found for Parameter Name {0}

DSE3574=All column selected.

DSE3575=InComparable DataType {0}

DSE3576=Column {0} used in NATURAL join cannot have qualifier

DSE3577=CROSS JOIN IS ONLY POSSIBLE

DSE3802=Not found.

DSE3804=Npe from variable column.

DSE4104=Error converting data type varchar to float.

DSE4106=Cannot insert in {0} data type value.

DSE4107=Invalid data type {0} is passed for argument {1} in function {2}.

DSE4108=Invalid data type {0} is passed in function {1}.

DSE4109=Invalid operator for data type. operator EQUALS minus, type EQUALS {0}.

DSE4111=The {0} aggregate operation cannot take a {1} data type as an argument.

DSE4112=Syntax error converting the {0} value {1} to a column of data type {2}.

DSE4113=Datatype has been set to {0} type for decimal value.

DSE4114=The BLOB,CLOB data types cannot be compared or sorted, except when using is null operator.

DSE4115=Invalid column name {0}.

DSE4116=Iterator not aligned to any valid location. (First call beforefirst() or first()).

DSE4117=Iterator not aligned to any valid location. (First call afterlast() or last()).

DSE4118=Object do not belong to supported datatypes.

DSE4119=The {0} requires one argument.

DSE4120=Object is an instance of ignorevalue.

DSE4121=Syntax error converting the {0} value {1} to a column of data type {2} and value {3}.

DSE4122=Cannot perform an aggregate function on an expression containing an aggregate or a subquery.

DSE4123=Cannot Move to the key = {0}

DSE4124=Iterator is not Initialized.

DSE4125=Column Not Found.

DSE4126=Method not supported.

DSE4127=There are no Childs for Class -> {0}.

DSE5001=User Defined Function {0} not supported

DSE5002=Invalid Grantee in Grant/Revoke Statement

DSE5003=Invalid Event (Listener fired with SELECT Type Event with Operation Type update)

DSE5004=Invalid Operation Type in Event

DSE5005=Can't specify autoIncrement for more than one column

DSE5006=Can't specify default option for autoIncrement column

DSE5007=Invalid data type for autoIncrement

DSE5008=There are no primary or candidate keys in the referenced table {0} that match the referencing column list in the foreign key {1}.

DSE5009=No primary or unique constraint present

DSE5010=Schema contains some Database Objects i.e. tables, views, Domains, Routines, Triggers etc.
Can not be dropped

DSE5011=Invalid user name {0}

DSE5012=Cannot have more than one null call clause

DSE5013=Cannot have more than one transformgroupspecification

DSE5014=Routines does not support multiple privileges

DSE5015=Not a valid view both materialized and INTO tablename should be specified

DSE5016=Invalid catalog name for table

DSE5017=Datatypedescriptor and columns table result mismatch

DSE5018=Contains columns refering domain

DSE5019=Current user is not authorized to drop routine

DSE5020=Record to be deleted not present

DSE5021=Language other than SQL is not supported

DSE5022=Null-Call clause shall not be specified in case of SQL invoked procedures

DSE5023=No owner exist for Schema {0}

DSE5024=Procedure Exist for same name and same number of parameters

DSE5025=Catalog/Schema of routine name and specific name should be same

DSE5026={0} has no references rights on table {1}

DSE5027=Result Set is Closed

DSE5028=View {0} is not a materializedview.

DSE5029=Problem in case of delete event on RecordSetBufer {0}.

DSE5030=Problem in case of insert event on RecordSetBufer {0}.

DSE5031=Insert not allowed for query {0}.

DSE5032=Error Occured while setting values for Record {0}.

DSE5034=Column {0} is not updatable in query {1}.

DSE5036=There is already one record in update state.

DSE5037=Deleterow cannot be called for record yet to be inserted.

DSE5038=Loadrecordforidentity cannot be called for record yet to be inserted.

DSE5040=Ignore Values cannot be passed in this sequence for client parameters.

DSE5041=Number of Parameter Infos for client parameters {0} are more than parameters in query {1}.

DSE5042=Autonumber value cannot exceed {0}.

DSE5045=Invalid event Type {0}.

DSE5046=Role Dependency graph is not initialized properly.

DSE5047=Privilege Dependency graph is not initialized properly.

DSE5101=Cannot drop view {0}, because this view is a materailized view.

DSE5102=Cannot drop materailized view {0}, because this view is not a materailized view.

DSE5502=Invalid database name {0}.

DSE5503=You can not change the ISOLATION LEVEL.

DSE5504=SEQUENCE {0}.NEXTVAL exceeds maxvalue and cannot be instantiated.

DSE5505=SEQUENCE {0}.NEXTVAL goes below minvalue and cannot be instantiated.

DSE5506=Sequence already exists {0}.

DSE5507=Duplicate or conflicting {0} specifications.

DSE5508=Increment must be a non-zero integer.

DSE5509=Minvalue cannot be less than {0}.

DSE5510=MAXVALUE Can not be greater than {0}

DSE5511=Order not Supported

DSE5512=MINVALUE must be less than MAXVALUE

DSE5513=START WITH should lie between MINVALUE and MAXVALUE

DSE5514=Absoulte of the INCREMENT value must be less than or equal to MAXVALUE minus MINVALUE and Should not be "0"

DSE5515=MAXVALUE cannot be made to be less than the current value

DSE5516=Sequence {0} does not exist

DSE5517=MINVALUE cannot be made greater than the current value

DSE5518=Couldn't Move to Keys Specified.

DSE5519=Database {0} already connected.

DSE5520=Date {0} should lie between {1} and {2}.

DSE5521=Pointer is not set after referesh.

DSE5522=Database is in use.

DSE5523=Cannot create directory on specified path.

DSE5524=Remove ChildServerSession.

DSE5525=START WITH cannot be less than minvalue.

DSE5526=START WITH cannot be greater than maxvalue.

DSE5528=No record found for keys specified.

DSE5529=U cannot get parentsessionid without starting any save point.

DSE5530=Sessionid list contains only one element.No Start save point exists.

DSE5531=SessionID's are not hidden. Call hideSavepoint first.

DSE5532=StartSavePoint NOT Started or no of start savepoints started are less than 2.

DSE5533=Allow parallelsavpoint NOT started yet.

DSE5534=Commit last savepoint first.

DSE5535=Ignore parallel savepoint first.

DSE5536=No element removed from list.

DSE5537=Invalid instance of iterator {0}.

DSE5538=No referenced table found for column {0}.

DSE5539=Databasefile {0} is either removed or deleted from path.

DSE5540=Invalid constraint name or cannot defer a constraint that is not defferable.

DSE5541=Invalid constraint mode {0}.

DSE5542=Can not specify {0} more than one time.

DSE5543=System Database is not properly created, delete the directory from the path first.

DSE5544=UserDatabase is not properly created,drop the database first.

DSE5545=Value {0} is out of range of {1} data type.

DSE5546=Value larger than specified precision allows for this column.

DSE5547=Data type {0} is not convertible into data type {1}.

DSE5548=Column name {0} appears more than once in the result column list.

DSE5549=Invalid date-time result.

DSE5551=XML file at specified Path {0} not found.

DSE5552=Blob data file at specified Path {0} not found.

DSE5553=Clob data file at specified Path {0} not found.

DSE5554=XML File Write Exception.

DSE5555=Blob data file Write Exception.

DSE5556=Clob data file Write Exception.

DSE5557=Error while getting LOB data.

DSE5558=Database is in Read Only Mode.

DSE5560=Schedule {0} to be dropped does not exist.

DSE5561=Schedule {0} already exists.

DSE5562=Database {0} for which scheduler is added does not exist.

DSE5564=Database to be backed up is in use.

DSE5566=Database {0} to be restored already exists.

DSE5567=Database {0} to be backed up already exists.

DSE5568=Database {0} can not be restored.

DSE5569=Database can not be restored with name {0}.

DSE5570=Database {0} can not be backed up.

DSE5571=Database can not be backed up with name {0}.

DSE5572=Invalid path {0}.

DSE5573=SystemDatabase and database {0} not compatible.

DSE5574=Can't take backup as Source Path And Destination Path are same.

DSE5575=User {0} is not having privilege to take Backup.

DSE5576=Database name can not be {0}.

DSE5577=Cannot get Connection as Backup is under process.

DSE5578=Adding Schedule is not supported on this version.

DSE5579=Backup corrected - some files are either removed or deleted from path - Start backup after removing all files from the path.

DSE5581=Access denied. Save point already started.

DSE5582=Exception: Trigger in recursion exceeding count 16.

DSE5583=Exception: Statement Trigger in recursion exceeding count 16.

DSE5584= Table has been already dropped.

DSE5590= Feature{0} not supported below Version3.0

DSE5591= Version incompatible as executable jar doesn't support Backward Compatibility

DSE6001={0} is non-comparable data type, cannot be used in Distinct/Predicates/Union/Intersect queries.

DSE6004=Invalid type {0} in Like predicate.

DSE6005=Having Clause should not contain ContainsClause

DSE6006=ORDER BY columns must appear in the select list if DISTINCT is contained in Select List.

DSE6007=Expression {0} cannot be specified in search condition.

DSE6008=Sequence number not allowed here.

DSE6009=SubQuery is not allowed in order by clause.

DSE6010=FullTextIndex name should be given if index exists on multiple column.

DSE6011=Contains clause not supported for queries involving more than one table or view.

DSE6012=Select Query is not supported in select list.

DSE6013=Sequence is not allowed in Group by and order by clause

DSE7001=Foreign key {0} has implicit reference to object {1} which does not have a primary key defined on it.

DSE7002=Number of referencing columns in foreign key differs from number of referenced columns, table {0}.

DSE7003=Foreign key {0} references invalid table {1}.

DSE7005=Invalid column(s) {0} specified in CHECK constraint {1}.

DSE7006=Foreign key {0} references invalid column {1} in referenced table {2}.

DSE7007=Foreign key {0} references invalid column {1} in referencing table {2}.

DSE7008=Cannot alter table {0} because this table does not exist in database.

DSE7009=Cannot drop the table {0}, because it does not exist in the database.

DSE7010=Cannot drop the view {0}, because it does not exist in the database.

DSE7051=[clientstatement]you cannot SET client parameters for the query {0}.

DSE7052=[rsb]one record is already taken for insertion FIRST COMMIT that and then try again.

DSE7053=[rsb]Invalid call.

DSE7054=[rsb]you cannot call update row on inserted record.

DSE7055=[rsb]there is already one record in UPDATE state.

DSE7056=[rsb]this record was not updated.

DSE7057=Cannot drop the index {0} from table {1}, because it does not exist in the database.

DSE7058=User {0} already exists in the database.

DSE7059=Cannot drop the user {0}, because it does not exist.

DSE7060=Cannot create an index on {0}, because this table does not exist in database.

DSE7061=Udt support not available.

DSE7062=Duplicate username listed.

DSE7063=Invalid interval {0}.

DSE7064=Column count cannot be greater than one.

DSE7065=Cannot drop the trigger {0}, because it does not exist in the database.

DSE7066=Invalid grantee in grant statement.

DSE7067=Option {0} can be defined at most once.

DSE7068=HAS RECORD cannot be used in Group by clause.

DSE7069=Illegal Mapping.

DSE7070=Count cannot be less than or equal to zero, specified count is {0}.

DSE7071=Chained table info for table {0} not found.

DSE7072=Table details mapping does not contain table details {0}.

DSE7073=Table {0} already has a primary key defined on it.

DSE7074=More than one key specified in column level FOREIGN KEY constraint, table {0}.

DSE7075=ALTER TABLE ADD CONSTRAINT statement conflicted with FOREIGN KEY constraint {0}. The conflict occurred in table {1}.

DSE7076=Catalog {0} does not exist.

DSE7077=Column names in each view must be unique. Column name {0} in view {1} is specified more than once.

DSE7078=ALTER TABLE DROP COLUMN statement conflicted with FOREIGN KEY constraint {0}. The conflict occurred in table {1}, column {2}.

DSE7079=ALTER TABLE ALTER COLUMN SET DEFAULT statement failed because column {0} does not exist in table {1}.

DSE7080=ALTER TABLE ALTER COLUMN DROP DEFAULT statement failed because column {0} does not exist in table {1}.

DSE7081=ALTER TABLE DROP COLUMN statement conflicted with VIEW {0}. The conflict occurred in table {1}, column {2}.

DSE7082=ALTER TABLE DROP COLUMN statement conflicted with TRIGGER {0}. The conflict occurred in table {1}, column {2}.

DSE7083=ALTER TABLE DROP COLUMN statement conflicted with CHECK constraint {0}. The conflict occurred in table {1}, column {2}.

DSE7084=ALTER TABLE ADD CONSTRAINT statement conflicted with UNIQUE constraint {0}. The conflict occurred in table {1}.

DSE7085=ALTER TABLE ADD CONSTRAINT statement conflicted with PRIMARY KEY constraint {0}. The conflict occurred in table {1}.

DSE7086=ALTER TABLE ADD CONSTRAINT statement conflicted with CHECK constraint {0}. The conflict occurred in table {1}.

DSE7087=ALTER TABLE ADD COLUMN statement conflicted with PRIMARY KEY constraint {0}. ALTER TABLE only allows columns to be added that can contain nulls or have a DEFAULT definition specified. Column {1} cannot be added to table {2} because it does not allow nulls and does not specify a DEFAULT definition.

KEY constraint {0}. The conflict occurred in table {1}.

DSE7088=ALTER TABLE ADD COLUMN statement conflicted with UNIQUE KEY constraint {0}. The conflict occurred in table {1}.

DSE7089=ALTER TABLE ADD COLUMN statement conflicted with CHECK CONSTRAINT {0}. ALTER TABLE only allows columns to be added that can contain nulls or have a DEFAULT definition specified. Column {1} cannot be added to table {2} because it does not allow nulls and does not specify a DEFAULT definition.

DSE7090=Database {0} specified is either removed or deleted from path.

DSE7091=HAS RECORD ColumnDetail contains invalid table name(s) {0}.

DSE7092=Duplicate column name {0}.

DSE7093=Alias Name Should Be specified with Aggregate/Expressional or Scalar Functional columns.

DSE7094=Invalid boolean type {0}.

DSE7095=ColumnNames specified are not equal in length.

DSE7096=Node size should be greater than 2.

DSE7097=Invalid Instance Of Iterator {0}

DSE7098=Invalid status in Top Iterator

DSE7099=Iterator for table not found

DSE8000=Position not set

DSE8001=Existing value and delete event fired

DSE8002=Specified System table {0} not found

DSE8003=Table details mismatch

DSE8004=Column name for index passed {0} is not Found

DSE8005=Index not found of reference {0} in references {1}

DSE8006=Invalid join type {0}

DSE8007=Query passed is null.

DSE8008=Column {0} is not a functional column.

DSE8009=Invalid number

DSE8010=DROP TABLE statement conflicted with VIEW {0}. The conflict occurred in table {1}.

DSE8011=DROP TABLE statement conflicted with TRIGGER {0}. The conflict occurred in table {1}.

DSE8012=DROP TABLE statement conflicted with CHECK constraint {0}. The conflict occurred in table {1}.

DSE8013=DROP TABLE statement conflicted with FOREIGN KEY constraint {0}. The conflict occurred in table {1}.

DSE8014=DROP VIEW statement conflicted with VIEW {0}. The conflict occurred in view {1}.

DSE8015=DROP VIEW statement conflicted with TRIGGER {0}. The conflict occurred in view {1}.

DSE8016=Inconsistent data type, value should be characterstringliteral

DSE8017="DATE" Keyword must specify in default clause for column {0}

DSE8018=Inconsistent data type, value should be booleanliteral for column {0}

DSE8019=Default value is too large for column {0}

DSE8021=Method {0} is Wrongly Called For SelectedColumnIterator

DSE8022=Invalid timestamp data type value. Timestamp format must be yyyy-mm-dd hh:mm:ss.fff.

DSE8023=Invalid time data type value. Time format must be hh:mm:ss.

DSE8024=Invalid date data type value. Date format must be yyyy-mm-dd.

DSE8025=Acess Denied Do not have Select privileges on column {0} of table {1}

DSE8026=Invalid or Parameterised queries

DSE8027=Invalid column(s) {0} specified in trigger condition for trigger {1}

DSE8028=Invalid column(s) {0} specified in trigger statement for trigger {1}

DSE8029=Invalid default clause for data type {0}

DSE8030=Object type {0} is not compatible with privilege type {1}

DSE8031=Row and table simuntaneoulsy can not be present in trigger definition

DSE8032=Old or new values alias list can not be specified for Statement trigger

DSE8033=Invalid country code {0}

DSE8034=Invalid Language code {0}

DSE8035=Loop statement does not contain terminating statement

DSE8036=Beginning label should be specified in case ending label used

DSE8037=Beginning label and ending label used should match for a statement

DSE8038=Invalid label used in leave or iterate statement

DSE8039=Duplicate variable declaration

DSE8040=Cursor {0} already closed

DSE8041=No value found for parameter {0}

DSE8042=Only prepared statement like functionality supported in case of statements other than CALL statement

DSE8043=Only NEXT operation allowed on non scrollable curosr

DSE8044=Cardinality mismatch between query specification and fetch target list

DSE8045=Invalid Variable name in fetch target list

DSE8046=Cursor in non updatable

DSE8047=Table name {0} not present in query specification of cursor {1}

DSE8047=Variable {0} not declared in procedure {1}

DSE8049=The maximum size for all index columns can not exceed {0} bytes. The index {1} with size {2} bytes can not be created.

DSE8050=such column list already indexed.

DSE8051=Cursor {0} either not opened or has been closed

DSE8052=Either EXECUTE not called or command does not produce result set

DSE8053=Either URL or dataSourceName must be specified to create connection

DSE8054=No such DataSource {0}

DSE8055=Data Source name can not be null

DSE8056=Transaction Isolation Level passed is Not Valid

DSE8057=Invalid ResultSetType {0}

DSE8058=Invalid concurrency {0}

DSE8059=UDTs are not supported

DSE8060=Parameter Index can not be less than 1,passed parameter index is {0}

DSE8061=Not A Valid Direction {0}

DSE8062=In case of type Forward only ,only valid direction is fetch forward

DSE8063=Fetch Size can not be less than 0

DSE8064=UpdateRow called while curosr is on newly insert row

DSE8065=Invalid cursor position

DSE8066=Before Calling insertRow ,cursor must be on insertrow

DSE8067=Auto column value can not be NULL

DSE8068=Do not have rights to revoke privileges on object {0}

DSE8069=length/precision/largeobjectlength can not zero for datatypes

DSE8090=Cursor statements cannot be executed through Daffodil DB Shell/Daffodil DB Browser.

DSE8091=Invalid column Indexes for GeneratedKeys

DSE8092=Invalid role {0}.

DSE8093=The role can not be granted to itself or any of its applicable roles

DSE8094=Invalid role/user {0}.

DSE8095=No such role authorization descriptor found to delete.

DSE8096=Role to be granted does not lie in applicable role of grantor {0} or doesn't have WITH ADMIN OPTION.

DSE8097=Currently active User {0} can not be dropped

DSE8098=Role {0} is currently active/Lies in applicable roles.

DSE8100=Incompatible data type {0} and {1}.

DSE8101=Syntax error converting data type {0} to {1}.

DSE8102=Cannot create index on view

DSE8103=object can not be created with name greater than 128 characters

DSE8104=System or Administrator user can not be dropped

DSE8105=Do not have CREATE/DROP user permission.

DSE8106=Syntax error converting {0} data type to date data type.

DSE8107=Length {0} exceeds premissable values for length or is invalid.

DSE8108=Invalid check constraint condition {0}

DSE8109=Invalid column name/references in triggered action

DSE8110=Do not have DROP permission on {0}

DSE8111=Invalid start index passed to the {0} function. Start index can not be less than one.

DSE8112=Invalid length parameter passed to the {0} function. Length can not be negative.

DSE8113=repeat counter too large.

DSE8114=Invalid counter passed to the {0} function. Counter can not be negative.

DSE8116=Cannot alter domain {0} because this domain does not exist in database.

DSE8115=can not add column as check constraint applied is violated.

DSE8117= {0} length can not exceed {1}

DSE8118=Problem in creating file {0} either it contains illegal characters or blank spaces

DSE8119=Invalid columnName or indexName used in ContainClause{0}

DSE6002=Blank or Stop words cannot be used for searching.

DSE6003=For Update Option cannot be used if query contains multiple tables, view, group by or having clause.

DSE8120=Specified class is not valid {0}

DSE8121=wrong name: {0}

DSE8122=Specified jar name is not valid {0}

DSE8123=Number of parameters in procedure declaration does not match with number of parameters in java method

DSE8124=Datatype mismatch in parameters at {0}

DSE8125=Data Type not supported

DSE8126=Method {0} does not exist in {1}

DSE8127=Class does not have <init>(java.sql.Connection) constructor

DSE8128 =Contains Clause of the query contained only ignored words

DSE8129={0} must be the current user.

DSE8131=Role {0} not granted to user {1}

DSE8132=Do not have sufficient privileges for select

DSE8133=CurrentUser/CurrentRole does not have execute privileges on {0}.

DSE8134=Incorrect alias {0}

DSE8135=Alias name not provided for column {0}

DSE8136=This Feature not supported in version {0}

DSE8137=Do not have sufficient privileges for create schema with {0} catalog.

DSE8138=Contain clause not supported in searched condition.

DSE8139={0}

DSE8141=Number of arguments exceed actual number of columns

DSE8142=Parameterized statement not allowed in triggers.

DSE8143=User/Role {0} has Insufficient privileges.

DSE8144=Length of column {0} should be equal to 1031.

DSE8145=Error while executing {0} method

DSE8146="TIME" Keyword must specify in default clause for column {0}

DSE8147="TIMESTAMP" Keyword must specify in default clause for column {0}

DSE8148=Value larger than specified precision not allowed for column {0}

DSE8149=Data type value is invalid for column {0} / Date format must be yyyy-mm-dd.

DSE8150=length/precision/largeobjectlength of datatype can not zero for column {0}

DSE1186=Cursor {0} is not scrollable

DSE8151=Do not have CREATE/DROP database permission for {0} session.

DSE8152=Contains clause not allowed in view definition.

DSE8153={0} in triggered action for trigger {1}

DSE8154=More than one primary key constraint can not be applied on column {0}

DSE8155=More than one unique key constraint can not be applied on column {0}

DSE8156=More than one not null constraint can not be applied on column {0}

DSE8157=Both primary key and unique key constraint's can not be applied on column {0}

DSE8158=Column {0} should exist on search condition

DSE8159=Parameters statement not allowed in View {0}

DSE8160=Exception: Procedure in recursion exceeding count 32

DSE8161=Unique/Primary key constraint already exist on table {0}

DSE8162=Query expression's length {0} can not exceed "4192" characters for view {1}.

DSE8163=ALTER TABLE DROP COLUMN statement conflicted with PRIMARY/UNIQUE/FOREIGN KEY constraint {0}. The conflict occurred in table {1}, column {2}.

DSE8164=Parameterised query not allowed in Stored Procedure.

DSE8165=Default value's length not supported up to "1024".

DSE8166=Current session does not have rights on schema {0}

DSE8167=Grantor {0} does not have any privilege with grant option on object {1}.

DSE8168=Can not create default schema {0}.

DSE8169=Can not drop default schema {0}.

DSE8170=Result of repeat function exceeds the maximum length of char data type.

DSE8171=Constraint {0} should be less than 128 characters

DSE8172=Parameterized statement not allowed in domains.

DSE8173=Invalid parameter {0}.

DSE8174=Aggregate function not allowed in procedure call.

DSE8175=ColumnName/VariableName {0} is invalid/not declared.

DSE8176=Current user {0} must have references rights on column {1} of table {2}.

DSE8177=The value you entered is not consistent with the data type or length of the parameter.

DSE8178=Datatype {0} should be predefined datatype.

DSE8179=User {0} must be the database owner OR admin user.

DSE8180=Database can not be created with user {0}.

DSE8181=Database owner {0} can not be dropped.

DSE8182=Procedure {0} does not exist in databse.

DSE8183=Can not declare variable {0} with same name.

DSE8184=SubQuery not allowed in procedure parameters.

DSE8185=Role {0} does not exist in database.

DSE8186=Trigger table alias does not match with alias {0} used in when condition.

DSE8187=Can not add multiple unique OR primary key constraint on same column in table {0}

DSE8188=Acess Denied Do not have Usage privileges on domain {0}

DSE8189=Schema statements(Create, Alter, And Drop) not allowed in procedures.

2. Country Code Table

Country Name	Country Code	Country Name	Country Code
Andorra	AD	Botswana	BW
United Arab Emirates	AE	Belarus	BY
Afghanistan	AF	Belize	BZ
AG	AG	Canada	CA
Anguilla	AI	CC	CC
		Central African	
Albania	AL	Republic	CF
Armenia	AM	Congo	CG
Netherlands Antilles	AN	Switzerland	CH
Angola	AO	Côte d'Ivoire	CI
AQ	AQ	CK	CK
Argentina	AR	Chile	CL
AS	AS	Cameroon	CM
Austria	AT	China	CN
Australia	AU	Colombia	CO
Aruba	AW	Costa Rica	CR
Azerbaijan	AZ	Cuba	CU
Bosnia and			
Herzegovina	BA	Cape Verde	CV
Barbados	BB	CX	CX
Bangladesh	BD	Cyprus	CY
Belgium	BE	Czech Republic	CZ
Burkina Faso	BF	Germany	DE
Bulgaria	BG	Djibouti	DJ
Bahrain	BH	Denmark	DK
Burundi	BI	Dominica	DM
Benin	BJ	Dominican Republic	DO
Bermuda	BM	Algeria	DZ
Brunei	BN	Ecuador	EC
Bolivia	BO	Estonia	EE
Brazil	BR	Egypt	EG
Bahamas	BS	Western Sahara	EH
Bhutan	BT	Eritrea	ER
BV	BV	Spain	ES

Country Name	Country Code	Country Name	Country Code
Ethiopia	ET	Indonesia	ID
Finland	FI	Ireland	IE
Fiji	FJ	Israel	IL
FK	FK	India	IN
Micronesia	FM	IO	IO
FO	FO	Iraq	IQ
France	FR	Iran	IR
FX	FX	Iceland	IS
Gabon	GA	Italy	IT
United Kingdom	GB	Jamaica	JM
GD	GD	Jordan	JO
Georgia	GE	Japan	JP
French Guiana	GF	Kenya	KE
Ghana	GH	Kyrgyzstan	KG
GI	GI	Cambodia	KH
GL	GL	Kiribati	KI
Gambia	GM	Comoros	KM
Guinea	GN	KN	KN
Guadeloupe	GP	North Korea	KP
Equatorial			
Guinea	GQ	South Korea	KR
Greece	GR	Kuwait	KW
GS	GS	KY	KY
Guatemala	GT	Kazakhstan	KZ
GU	GU	Laos	LA
Guinea-Bissau	GW	Lebanon	LB
Guyana	GY	LC	LC
Hong Kong	HK	Liechtenstein	LI
HM	HM	Sri Lanka	LK
Honduras	HN	Liberia	LR
Croatia	HR	Lesotho	LS
Haiti	HT	Lithuania	LT
Hungary	HU	Luxembourg	LU

Country Name	Country Code	Country Name	Country Code
Latvia	LV	NR	NR
Libya	LY	Niue	NU
Morocco	MA	New Zealand	NZ
Monaco	MC	Oman	OM
Moldova	MD	Panama	PA
Madagascar	MG	Peru	PE
MH	MH	French Polynesia	PF
		Papua New	
Macedonia	MK	Guinea	PG
Mali	ML	Philippines	PH
Myanmar	MM	Pakistan	PK
Mongolia	MN	Poland	PL
MO	MO	PM	PM
MP	MP	PN	PN
Martinique	MQ	Puerto Rico	PR
Mauritania	MR	Portugal	PT
Montserrat	MS	PW	PW
Malta	MT	Paraguay	PY
Mauritius	MU	Qatar	QA
MV	MV	RE	RE
MW	MW	Romania	RO
Mexico	MX	Russia	RU
Malaysia	MY	Rwanda	RW
Mozambique	MZ	Saudi Arabia	SA
Namibia	NA	SB	SB
New Caledonia	NC	Seychelles	SC
Niger	NE	Sudan	SD
NF	NF	Sweden	SE
Nigeria	NG	Singapore	SG
Nicaragua	NI	SH	SH
Netherlands	NL	Slovenia	SI
Norway	NO	SJ	SJ
Nepal	NP	Slovakia	SK

Country Name	Country Code	Country Name	Country Name
Slovakia	SK	Uganda	UG
Sierra Leone	SL	UM	UM
SM	SM	United States	US
Senegal	SN	Uruguay	UY
Somalia	SO	Uzbekistan	UZ
Suriname	SR	Vatican	VA
ST	ST	VC	VC
El Salvador	SV	Venezuela	VE
Syria	SY	British Virgin Islands	VG
Swaziland	SZ	U.S. Virgin Islands	VI
TC	TC	Vietnam	VN
Chad	TD	Vanuatu	VU
French Southern Territories	TF	WF	WF
Togo	TG	WS	WS
Thailand	TH	Yemen	YE
Tajikistan	TJ	Mayotte	YT
Tokelau	TK	Yugoslavia	YU
Turkmenistan	TM	South Africa	ZA
Tunisia	TN	Zambia	ZM
Tonga	TO	Zaire	ZR
East Timor	TP	Zimbabwe	ZW
Turkey	TR		
Trinidad and Tobago	TT		
TV	TV		
Taiwan	TW		
Tanzania	TZ		
Ukraine	UA		

3. Language Code Table

Language Name	Language Code	Language Name	Language Code
Afar	aa	French	fr
Abkhazian	ab	Frisian	fy
Afrikaans	af	Irish	ga
Amharic	am	Scots Gaelic	gd
Arabic	ar	Galician	gl
Assamese	as	Guarani	gn
Aymara	ay	Gujarati	gu
Azerbaijani	az	Hausa	ha
Bashkir	ba	Hebrew	he
Byelorussian	be	Hindi	hi
Bulgarian	bg	Croatian	hr
Bihari	bh	Hungarian	hu
Bislama	bi	Armenian	hy
Bengali	bn	Interlingua	ia
Tibetan	bo	Indonesian	id
Breton	br	Interlingue	ie
Catalan	ca	Inupiak	ik
Corsican	co	Indonesian	in
Czech	cs	Icelandic	is
Welsh	cy	Italian	it
Danish	da	Inuktitut	iu
German	de	Hebrew	iw
Bhutani	dz	Japanese	ja
Greek	el	Yiddish	ji
English	en	Javanese	jw
Esperanto	eo	Georgian	ka
Spanish	es	Kazakh	kk
Estonian	et	Greenlandic	kl
Basque	eu	Cambodian	km
Persian	fa	Kannada	kn
Finnish	fi	Korean	ko
Fiji	fj	Kashmiri	ks
Faroese	fo	Kurdish	ku

Language Name	Language Code	Language Name	Language Code
Kirghiz	ky	Slovak	sk
Latin	la	Slovenian	sl
Lingala	ln	Samoa	sm
Laothian	lo	Shona	sn
Lithuanian	lt	Somali	so
Latvian (Lettish)	lv	Albanian	sq
Malagasy	mg	Serbian	sr
Maori	mi	Siswati	ss
Macedonian	mk	Sesotho	st
Malayalam	ml	Sundanese	su
Mongolian	mn	Swedish	sv
Moldavian	mo	Swahili	sw
Marathi	mr	Tamil	ta
Malay	ms	Telugu	te
Maltese	mt	Tajik	tg
Burmese	my	Thai	th
Nauru	na	Tigrinya	ti
Nepali	ne	Turkmen	tk
Dutch	nl	Tagalog	tl
Norwegian	no	Setswana	tn
Occitan	oc	Tonga	to
Oromo (Afan)	om	Turkish	tr
Oriya	or	Tsonga	ts
Punjabi	pa	Tatar	tt
Polish	pl	Twi	tw
Pashto (Pushto)	ps	Uighur	ug
Portuguese	pt	Ukrainian	uk
Quechua	qu	Urdu	ur
Rhaeto-Romance	rm	Uzbek	uz
Kirundi	rn	Vietnamese	vi
Romanian	ro	Volapuk	vo
Russian	ru	Wolof	wo
Kinyarwanda	rw	Xhosa	xh
Sanskrit	sa	Yiddish	yi
Sindhi	sd	Yoruba	yo
Sangho	sg	Zhuang	za
Serbo-Croatian	sh	Chinese	zh
Sinhalese	si	Zulu	zu