ORACLE

HISTORY OF ORACLE

Larry Elison, Edoats, Bobmyron formed a company called RSI (Relational Software Incorporated) in the year 1977. This company built a RDBMS called ORACLE. In 1977 RSI released ORACLE version 1.0 is a prototype.

In 1979, RSI delivered its first product to customers. The ORACLE RDBMS Version 2 worked on the digital PDP-11, running on the RSX-11 operating system and was soon ported to the decvax systems.

In 1983, the release of version 3 which made changes to the SQL Language as well as performance enhancement and other improvements. At this point RSI changed its name to ORACLE Corporation.

ORACLE version 4 was released in the year 1984. This version supported both decvax systems and IBM operating systems.

ORACLE version 5 was introduced in 1985 was a milestone because it introduced the client/server computing to the market with the use of SQL*Plus. Version 5 was also the first MS-DOS product.

In 1988, ORACLE version 6 introduced a variety of performance improvements, functionalities & enhancements. ORACLE version 7 released in 1992 included many architectural changes in the area of memory, CPU, Input, Output.

In 1997 ORACLE introduced ORACLE 8, which added object extensions.

Finally in 1999 ORACLE introduced ORACLE8i, which added Internet programming.

RDBMS Software: Sybase, Ingres, Informix, ORACLE.
Database Models: FMS, HDBMS, NDBMS, RDBMS.

FMS (File Management System)

FMS was the first database model used to store data in a computerized database. In this model the data item was stored sequentially in one large file.

Demerits

1. Queries are not supported.
2. Structure cannot be modified

HDBMS (Hierarchical Database Management System)

This model was introduced in the Information Management System (IMS) developed by IBMC in the year 1968. In this model data is stored in tree structure. This can be said to have a Parent – Child relationship.

Demerits

1. This model supports simple queries only.
2. Structure cannot be modified.

NDBMS (Network Database Management System)

This model was developed by a group called CODASYL (Conference On Data Systems Language). In this model data is stored in linked sets. This can be said to have Multiple Parent – Child relationship.

Demerits

1. It supports simple & complex queries.
2. Structure cannot be modified.

RDBMS (Relational Database Management System)

Relation : Is a double dimension view it consists of rows & columns, in other words it is called a TABLE.

Data : Data is a collection of facts, which is in the form of alphabets, Digits and other special characters.

Information : Processed data is known as information.

Database : Organized Collection of data or inter related collection of data, which is stored in computer system.

DBMS : It is a software, which is used to create, maintain, manage and manipulate data. Manipulate means UPDATION, INSERTION, DELETION, QUERY the data.

In the RDBMS model the data is stored in tables in the form of Row & column format.

Table : A group of similar records.

Row : Group of columns.

Column : Something that describes or qualifies.

Difference between SQL and SQL * Plus Commands

SQL	SQL * Plus
1. SQL commands must be terminated by a semicolon.	1. SQL * Plus commands may or may not be terminated by a semicolon.
2. SQL commands allows to type more than one line.	2. SQL * Plus commands must be typed in a single line.
3. SQL commands are stored in SQL Buffer.	3. SQL * Plus commands are not stored in Buffer.

DATA TYPES:

1. NUMERIC FAMILY:

a) **Number:** This data type allows fixed and floating-point numbers.

b) **Number (Size):** This data type allows fixed number of integer values.

c) **Number (Size, Dec):** This data type allows fixed number of integer and real values.

d) **Integer:** This data type allows only integer values. The default and max size is 38.

e) **Float:** This data type allows both integer and real values. The max and default size is 126 digits.

f) **Decimal[(SIZE)]:** This data type is same as number data type but it rounds the values.

II. CHARACTER or STRING FAMILY:

a) **CHAR[(Size)]:** This data type allows fixed length character data with spaces. Max size is 2000 characters. Default one character.

b) **VARCHAR2(Size):** This data type allows column length character data. The maximum size is 4000 characters.

c) **LONG:** This data type allows variable length character data upto 2 GB.

Restrictions of Long Data type:

1. A table can have only one long type column.
2. Long column's can't appear in integrity constraints. (except for null and not null constraints).
3. Long column can't be indexed.
4. A stored function cannot return a long value.

III. DATE:

This data type allows to insert dates only. The default ORACLE date format is DD-MON-YY (for ex: 07-JUL-00). The ORACLE valid date ranges from 1st JAN 4712 BC to 31st DEC 9999 AD.

IV. LONG RAW:

This data type allows to insert binary data (pictures and sounds…).

V. LOB (LARGE OBJECT) DATA TYPES:

Oracle 8 introduced a new data type LOB. It can store unstructured data such as text, image, video up to four giga bytes in size.

LOBs are similar to LONG and LONG RAW types, but differ in the following ways:

1)A table can have many number of LOBs.

2) The LOB locator is stored in the table column, either with or without the actual LOB value. LOB values can be stored in separate table spaces.

3) A LOB can be of up to 4gb in size.

4) LOBs permit efficient, random, piece-wise access to and manipulation of data.

Object Naming Rules:

1. The first character must be an alphabet. The remaining characters are digits,alphabets,underscore(_),$.
2. It must not be ORACLE keyword or reserved word.
3. It does not allow any blanks with in the table name.
4. It must not be an existed database object(table ,view, synonym,…) name.
5. The maximum length of table name is 30 characters long.

Database Objects

An Oracle database contains multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.

Table: Is the basic unit of the storage (Stores Data).
View: Logically represents subsets of data from one or more tables.
Synonym: Gives alternative names to objects(duplicate of table).
Index: Improves the performance of some queries.
Sequence: Generates primary key values (Sequence Numbers).

SQL & SQL * PLUS COMMANDS

CREATE TABLE

Category: DDLC TYPE: SQL

Syntax:

```
CREATE TABLE <TABLE NAME>
(COLUMN TYPE [NULL] [default expression],
COLUMN TYPE [NULL][default expression] - - -);
```

This command is used to create a new database table.

Note: A table can have minimum single column and maximum 1000 Columns.

Table Name : Is the name of the table. It must be satisfy all Object naming rules.

Column : Is the name of the column.

Type : Is the valid SQL data type.

Null : Null is a special keyword in ORACLE. It means unknown but it is not equal to zero and not equal to space.

Default : This clause is used to specify a default value.

Ex:

```
Create Table student
(
        rollno number(4),
        stu_name varchar2(15),
        stu_addr char(20),
        course char(15),
        join_date date,
        fees number(5),
        ipay number(5),
        due number(5)
);
```

DESCRIBE

TYPE: SQL * Plus

Syntax:

DESC[RIBE] < TABLE NAME> [;]

This command is used to display the definition of a specified database table.

Ex:

```
describe student;
    Or
desc student;
```

RENAME

TYPE: SQL

Syntax:

RENAME <OLD NAME> TO < NEW NAME>;

This command is used to change the table name from old to new.

Note: 1. The old table name must be an existed database table.
2. The new table name must not be an existed database table.

Ex: RENAME STUDENT TO STUD_INFO;

ALTER TABLE

CATEGORY: DDLC TYPE: SQL

Syntax:

ALTER TABLE <TABLE _ NAME>
ADD | MODIFY (COLUMN TYPE, - - -);

This command is used to change to the structure of a table.

ADD : This keyword is used to add a new column or columns into a table.
MODIFY : This keyword is used to change the data type or increasing and decreasing the column width.

Note: Before decreasing a column width the specified column should be null.
Ex: 1. ALTER TABLE STUD_INFO ADD(DIS NUMBER(2));

2. ALTER TABLE STUD_INFO MODIFY(ROLLNO NUMBER(6));

DROP TABLE

CATEGORY: DDLC TYPE: SQL

Syntax:

DROP TABLE <TABLE _ NAME >;

This command is used to remove the specified database table permanently from the current database.

Ex: DROP TABLE STUD_INFO;

INSERT — CATEGORY: DMLC TYPE: SQL

Syntax:

INSERT INTO <TABLE NAME >
[(COLUMN_NAME1, COLUMN_NAME2, - - -)] VALUES
(VALUE1,VALUE2,- - -);
This command is used to insert a new record into an existed database table.

Ex: INSERT INTO STUD_INFO(ROLLNO,STU_NAME)
VALUES(1001,'RAJESH');

MACRO SUBSTITUTION VARIABLES:

SQL * Plus SUPPORTS 2 MACRO VARIABLES &, &&

Syntax:

&<variable – name>
This operator defines a variable and it accepts different values.

Ex: INSERT INTO STUD_INFO(ROLLNO,STU_NAME)
VALUES(&ROLLNO,&STU_NAME);

SELECT — CATEGORY: DQLC OR DMLC TYPE: SQL

Syntax:

SELECT [ALL | DISTINCT] * | [TABLE_NAME.]
COLUMN_NAME [ALIAS], - - -, EXPRESSIONS
FROM <TABLE_NAME [<TABLE_NAME >]
[WHERE<CONDITION>]
[GROUP BY COLUMN_NAME [, COLUMN_NAME,.....]]
[HAVING <CONDITION>]
[ORDER BY COLUMN_NAME | POSITION [ASC | DESC]]
[UNION | UNION ALL | INTERSECT | MINUS]
[FOR UPDATE OF COLUMN_NAME [NOWAIT]];

This command is used for retrieving the information or necessary information from the specified database table or tables.

ALL	: All rows
DISTINCT	: It omits duplicates (rows or column values).
*	: All columns.

COLUMN_NAME : It is the name of the specified table column.
ALIAS : Duplicate name of the column.
EXPRESSIONS : arithmetic calculations.
TABLE_NAME : It is the name of the table.

Ex: 1) SELECT ALL * FROM STUD_INFO;
OR
SELECT * FROM STUD_INFO;
2) SELECT DISTINCT * FROM STUD_INFO;
3) SELECT DISTINCT NAME FROM STUD_INFO;
4) SELECT ROLLNO,COURSE FROM STUD_INFO;
5) SELECT ROLLNO STUDENT_ID,NAME FROM STUD_INFO;
6) SELECT ROLLNO,FEES,IPAY,FEES*20/100 DISCOUNT FROM STUD_INFO;

WHERE CLAUSE: Usually you do not want to retrieve all the rows in a table, particularly if a table has many rows. SQL provides a where clause in which you specify the criteria to be used for retrieving records. A where clause consists of one or more conditions that must be satisfied before a row is retrieved by the query.

OPERATORS:

1) **RELATIONAL OPERATORS:** >, >=, <, <=, =,! =
These operators are used to test the relation between two values.

2) **LOGICAL OPERATORS:** AND, OR, NOT
These operators combines the results of one or more expressions and it is called as logical expression. After testing the condition they return logical status (TRUE OR FALSE).

3) **SQL SPECIAL OPERATORS:**
IN, BETWEEN, LIKE, IS

SQL Special Operators:

IN

Syntax:

Column_Name [not] IN (value 1, - - -);

This operator defines a set in which a value may be existed or not. The set of values must be of the same type and each one is delimited by ",".

Ex: SELECT * FROM STUD_INFO
WHERE ROLLNO IN(1001,1002,1006);

SELECT * FROM STUD_INFO
WHERE NAME IN('VASU','vasu','Vasu');

BETWEEN

Syntax:

Column_Name [not] BETWEEN value1 and value2;

This operator is used to define a range in which the value may be existed or not.

Ex: SELECT * FROM STUD_INFO
WHERE FEES BETWEEN 3000 AND 5000;

LIKE

Syntax:

Column_Name [NOT] LIKE 'expression';

This operator is used only to test the character data. LIKE operator uses two wild card special characters.

1. %
2. _(under score)

%: It matches any string of characters including '0' number of characters.

Ex: SELECT * FROM STUD_INFO WHERE NAME LIKE 'A%';

_ (UnderScore): It matches only single character.

Ex: SELECT * FROM STUD_INFO WHERE NAME LIKE 'R_ _ _';

IS

Syntax:

Column_Name is [NOT] NULL;

This command is used to finding the null values.

Ex: SELECT * FROM STUD_INFO WHERE FEES IS NULL;

Create Table (Query Table) CATEGORY: DDLC TYPE: SQL

Syntax: Create Table <Table Name>
As query;

A second method to create a table is to apply the **as query** clause to both create the table and insert rows returned from the query.

Table Name: Is the name of the table.

Query: Is the select statement that defines the set of rows to be inserted into the new table.

Ex: CREATE TABLE STUD
AS SELECT * FROM STUD_INFO;

SELECT * FROM STUD;
SELECT * FROM STUD_INFO;

UPDATE CATEGORY: DMLC TYPE: SQL

Syntax:

UPDATE <Table_Name>
set Column_Name = Expression [,Column_Name =Expression]
[where <condition>];

This command is used to modifying an existed table Column values from old to new.

Ex: UPDATE TABLE STUD_INFO
SET FEES=6500 WHERE COURSE='DOA';

SELECT * FROM STUD_INFO;

DELETE Category: DMLC TYPE: SQL

Syntax:

DELETE [FROM] <Table_Name>
[Where <Condition>];

This command is used to delete a row or rows from the specified database table.

Ex: DELETE FROM STUD_INFO;
OR
DELETE STUD_INFO;

EX: DELETE FROM STUD_INFO WHERE ROLLNO=1001;

BUFFER MANAGEMENT COMMANDS

All buffer management commands are SQL * Plus commands. These commands are effects only on the buffer contents.

SQL is a single sentence language. The sentence once typed at the SQL prompt cannot be retrieved and corrected, if a spelling or syntax error was occurred. The technique used to overcome this constant retyping of sentences is to Create an ASCII FILE or buffer commands.

1. List 2. Change 3. Slash 4. Run 5. Append 6. Input
7. Delete 8. Save 9. Edit 10. Clear Buffer.

SQL * Plus Commands

1. GET 2. @ 3. HOST

LIST **Type : SQL * Plus**

Syntax: L [IST][M [N]][L[AST]]

This command is used to display all lines from the SQL buffer to the screen.

CHANGE **Type : SQL * Plus**

Syntax: C[HANGE]/OLD_TEXT/NEW_TEXT

This command is used to change the specified buffer line text from old to new.

SLASH **Type : SQL * Plus**

Syntax: /

This command is used to execute the current buffer contents.

RUN **Type : SQL * Plus**

Syntax: R[UN]

This command is used to retrieving and executing the current buffer contents.

APPEND **Type : SQL * Plus**

Syntax: A [PPEND] TEXT
This command is used to add a new text to the end of the current line of the current buffer.

DELETE **Type : SQL * Plus**

Syntax: DEL
This command is used to delete a current line of the current buffer.

INPUT **Type : SQL * Plus**

Syntax: I[NPUT] TEXT
This command is used to add a new line or lines after the end of the current line of the current buffer.

SAVE **Type : SQL * Plus**

Syntax: Save <Filename> [Create|Replace|Append]
This command is used to save the buffer contents permanently to a system file.

EDIT **Type : SQL * Plus**

Syntax: ED[IT] [FILENAME]
This command is used to open a SQL * Plus editor with the current buffer contents or with the specified system file contents.

CLEAR BUFFER **Type : SQL * Plus**

Syntax: CL[EAR] BUFF[ER]
This command is used to clear the current SQL buffer contents.

GET **Type : SQL * Plus**

Syntax: GET <Filename>[LIST | NOLIST]
This command is used to retrieving the specified file contents from file to the buffer and the screen.

@ **Type : SQL*Plus**

Syntax: @FILENAME

This command is used to retrieving and executing the specified file contents.

HOST **Type : SQL*Plus**

Syntax: HOST

This command is used to temporarily exit from the SQL * Plus to operating system prompt.

SQL FUNCTIONS

Function is a sub program or predefined program, which are meant to do certain task.

Functions are used to manipulate data items. They accept one or more arguments but it returns only one value. An argument is a user supplied constant or column reference.

Functions can be used to:

1. Perform calculations on data.
2. Modify individual data items.
3. Manipulate output for groups of rows.
4. Convert column data types.
5. Alter date formats for display.

Types of Functions:

1) Single Row Functions (scalar functions)
2) Group Row Functions (aggregate functions)

1. Single Row Functions

Functions that act on only one value at a time are called as Scalar or single row functions. A single row function returns one result for every row of a queried table or view.

Single row functions can be further grouped together by the data type of their arguments and return values.

1. Character or String functions
2. Numeric functions
3. Special functions

4. List functions
5. Date functions
6. Conversion functions

1. Character Functions:

These functions accept string or numeric data and it returns numeric or string data as output.

1. LOWER( ) 2. UPPER( ) 3. INITCAP( ) 4. CONCAT( ) 5. ASCII( )
6. CHR( ) 7. SUBSTR( ) 8. LENGTH( ) 9. INSTR( ) 10. REPLACE( )
11. TRANSLATE( ) 12. LTRIM( ) 13. RTRIM( ) 14. TRIM( ) 15. LPAD( )
16. RPAD( ) 17. SOUNDEX( )

DUAL: Is a system table it belongs to public . This table consists of one row and one column . The column name is DUMMY and the row is X.

1. LOWER( )

Syntax: LOWER (Column | String)

This function forces alpha character values, which are in upper or mixed case into lower case.

Ex: SELECT LOWER('WELcome') MES FROM DUAL;

2. UPPER( )

Syntax: UPPER(COLUMN | STRING)

This function forces alpha character values. Which are in lower or mixed case info UPPER case.

Ex: SELECT UPPER('welcome') MES FROM DUAL;

3. INITCAP( )

Syntax: INITCAP (column | String)

This function forces the first character of each word of the string or column to be capitalized, the remaining characters are displayed into lower case.

Ex: SELECT INITCAP('welcome to prs') MES FROM DUAL;

4. ASCII( )

Syntax: ASCII (Column | STRING|CHAR)

This function returns ASCII value of a given string or column.

Ex: SELECT ASCII('APPLE') ASC_VAL FROM DUAL;

Ex: SELECT ASCII('A') ASC_VAL FROM DUAL;

5. CHR( )

Syntax: CHR (N);

This function returns ASCII code of a given number.

Ex: SELECT CHR(65) ASC_CODE FROM DUAL;

Ex: SELECT CHR(97) ASC_CODE FROM DUAL;

6. SUBSTR( )

Syntax: SUBSTR (Column | String, pos[,n])

This function returns a portion of a string from the given string or Column, starting at the position POS to 'N' number of characters. If 'N' is omitted it returns the string starting at the position POS to the end of the string.

Ex: SELECT SUBSTR('WELCOME',4) NAME FROM DUAL;

Ex: SELECT SUBSTR('WELCOME',4,3) NAME FROM DUAL;

7. INSTR( )

(1) Syntax: INSTR (Column | String, Char)

This function returns the position number of the first occurrence of the given character in the Column or String.

Ex: SELECT INSTR('WELCOME','E') POS FROM DUAL;

(2) Syntax: INSTR (Column | String, Char, pos, N)

This function finds the character position of nth occurrence of a string in a column starting at the position POS.

Ex: SELECT INSTR('WELCOME','E',1,2) POS FROM DUAL;

8. LENGTH( )

Syntax: LENGTH (Column | String)

This function returns the length of a given String or Column.

NOTE: This function supports only varchar2 data type Columns.

Ex: SELECT LENGTH('WELCOME TO') NOC FROM DUAL;

9. REPLACE( )

Syntax: REPLACE (Column | String, search string,[replace_String]);

This function returns Column or Value with every occurrence of String replaced with replacement String. If replacement String is omitted, all occurrences of search String are removed.

Ex: SELECT REPLACE('WELCOME','COME','GO') NAME FROM DUAL;

10. TRANSLATE()

Syntax: TRANSLATE (Column | String, From, To)

This function translates column or string with FROM characters to the TO characters.

Ex: SELECT TRANSLATE('WELCOME','COME','GO') NAME FROM DUAL;

11. CONCAT()

Syntax: CONCAT (STRING 1,STRING 2)

Or

CONCAT (COLUMN 1, COLUMN 2)

This function returns the string 1 concatenated with string 2.

Ex: SELECT CONCAT('WEL','COME') NAME FROM DUAL;

Ex: SELECT CONCAT('RS.',120) PRICE FROM DUAL;

CONCATINATE OPERATOR:

Syntax: ||

This operator is used to concatenate more than 2 string expressions or numeric expressions.

Ex: SELECT 'LUX'||' RS.'||12 ITEM_PRICE FROM DUAL;

12. LTRIM()

Syntax: LRTIM (Column | String) char])

This function is used to remove the left leading blank spaces.

Ex: SELECT 120|| LTRIM(' RS') PRICE FROM DUAL;

13. RTRIM()

Syntax: RTRIM (Column | String [char])

This function is used to removes the trailing spaces or characters | from a given Column or String.

Ex: SELECT RTRIM('RS ') || 100 PRICE FROM DUAL;

14. TRIM()

Syntax: TRIM (Column | String)

This function is used to remove both the trailing and leading blank spaces from a given string or column.

Ex: SELECT 'LUX'||TRIM(' RS. ')||10 PRICE FROM DUAL;

15. LPAD()

Syntax: LPAD (Column | String, N [, char])

This function returns column or string, left padded to length, N with the sequence of characters in **char, if char** is omitted defaults to blanks. (Used in format report generation).

Ex: SELECT 100 ENO,LPAD('VASU',15,'.') NAME FROM DUAL;

16. RPAD()

Syntax: RPAD (Column | String, N_Exp, char)

This function is opposite to LPAD ().

Ex: SELECT RPAD(100,10,'.') ENO,'VASU' NAME FROM DUAL;

17. SOUNDEX()

Syntax: SOUNDEX (Column | String)

This function returns a Character String representing the sound of words for each Column.

Ex: SELECT * FROM STUD_INFO
WHERE SOUNDEX(NAME)=SOUNDEX('SUBBARAO');

2) Numeric Functions:

These functions accept numeric values and it returns the Output is numeric.

1) ABS( ) 2) CEIL( ) 3) FLOOR( ) 4) ROUND( ) 5) TRUNC( )
6) MOD( ) 7) POWER( ) 8) SQRT( ) 9) SIGN( )

1) ABS()

Syntax: ABS (Column | N)

This function finds the absolute value of a given column or N.

2) CEIL()

Syntax: CEIL (Column | N)

This function finds the smallest integer greater than or equal to the Column Or N.

3) FLOOR()

Syntax: FLOOR (Column | Value)

This function returns the largest integer <=given Column or value.

4) ROUND()

Syntax: ROUND (Column | Value [,n]

This function rounds the Column or Value to 'N' decimal values.

5) TRUNC()

Syntax: TRUNC (Column | Value [,n])

This function truncates the Column or Value to N decimal places.

6) MOD()

Syntax: MOD (M, N)

This function is used to find the remainder.

7) SQRT()

Syntax: SQRT (n)

This function is used to find the square root of a given number.

8) POWER()

Syntax: POWER (m,n)

This function returns the m to the Nth POWER Value.

9) SIGN()

Syntax: SIGN (Column | Value)

This function returns a numeric 1 or -1 or 0

3) LIST FUNCTIONS

1) GREATEST( ) 2) LEAST( )

1) GREATEST()

Syntax: GREATEST (value1, value2,. . .)

This function returns the greatest value in the given list of values.

EX: SELECT GREATEST(100,101,1000) BIG FROM DUAL;

2) LEAST()

Syntax: LEAST (Value1,Value2,. . .)

This function returns the least value in the given list of values.

EX: SELECT LEAST(100,101,1000) SMALL FROM DUAL;

4) SPECIAL FUNCTIONS

1) NVL()

Syntax: NVL (column | value, Expression)

This function is used to find the Null values. If the column value is null, it returns the given expression. If the column value is not null it returns the column value.

2) DECODE()

Syntax: Decode (Column | Value, Search_string1, Result1
[, search_string2, Result2,...][,default_value])

This function is a most powerful SQL function. This function Accept any type of data as input. This function is similar to IF command in other third generation languages.

Ex: SELECT DECODE(1,1,'ONE','INVALID') MES FROM DUAL;
SELECT DECODE('VASU','VASU','YES','vasu','yes','no') mes
From dual;

DATE FUNCTIONS

These functions operate on oracle Dates. All date functions accept date data type and it returns date or numeric values except months_between function, this function accepts date data and it returns number.

SYSDATE: It is a keyword that returns the current system date and time.

Arithmetic operators on Dates:

It is possible to perform calculations with dates using arithmetic operators + or –(addition & subtraction). We can add or subtract number constants as well as other dates from Dates.

OPERATIONS

1) **DATE +NUMBER:** Adds a no. of days to a date, producing a date.
2) **DATE - NUMBER:** Subtracts a no. of days from a date, producing a date.
3) **DATE - DATE:** Subtracts one date from another date producing a no. of days.

1) MONTHS_BETWEEN()

Syntax: MONTHS_BETWEEN (DATE1,DATE2)

This function returns the no. of months between date1 & date2.

2) ADD_MONTHS()

Syntax: ADD_MONTHS (DATE1,N)

This function adds 'N' no. of months to date1. N must be an integer and it can be –ve also.

3) LAST_DAY()

Syntax: LAST_DAY (DATE1)

This function returns the date of the last day of the given date month.

4) NEXT_DAY()

Syntax: NEXT_DAY (DATE1,'WEEK_NAME' | N)

This function returns the date of the next specified week.

6) CONVERSION FUNCTIONS

SQL provides no. of functions to control data type conversions. These functions convert a value from one data type to another.

1) TO_NUMBER()

Syntax: TO_NUMBER (CHAR_EXP)

This function converts char_exp, which contains a number to a number.

Ex: SELECT TO_NUMBER('100')*10 VAL FROM DUAL;

2) TO_CHAR()

Syntax1: TO_CHAR (N_Exp[,FMT])

This function converts numeric exp to a Character value in the given format specified by the FMT.

PICTURE	MEANING	EXAMPLE	OUTPUT
9	Numeric position	999	
	(no. of 9's	(123)	123
	Determines displayed width)	(2000)	###
$	Displays a floating $ sign	$999	
		(123)	123
		(1230)	###
.	Decimal point in the position	999.99	

	specified	(123)	123.00
		(1230)	###
	Comma in position specified	9,999	
		(123)	123
		(1200)	1,200
PR	Parenthesis -ve numbers	9999PR	
		(123)	123
		(-123)	<123>
MI	Minus sign to right	9999MI	
		(123)	123
		(-123)	123-

Ex: SELECT TO_CHAR(1000,'$9,999PR') SAL FROM DUAL;
SELECT TO_CHAR(-1000,'$9,999PR') SAL FROM DUAL;

2) TO_CHAR( )

Syntax2: To_Char (DATE [,FMT])

This function converts date expression to a character value in the format specified by the FMT.

Default Date format is DD-MON-YY

DATE FORMAT MODELS

1) DAY

D	: Day of Week (1 - 7)(1 – Sunday,2 – Monday,...)
DD	: Day of month (1 - 31)
DDD	: Day of the year (1 - 366)
DY	: Name of day, 3 letter abbreviation (MON, TUE,...)
DAY	: Name of the week (MONDAY, TUESDAY,...)

2) MONTH

MM	: Month Number (1 - 12)
MON	: Month Name, 3 letter abbreviation.
MONTH	: Name of the Month(JANUARY,...)

3) YEAR:

Y	: Last digit of the year
YY	: Last two digits of the year
YYY	: Last 3 digits of the year
YYYY	: Year Number
Y,YYY	: Year with comma in this position
YEAR	: Year spelled out
SYYYY	: 'S' prefixes B.C dates

4) WEEK:

W	: Week of Month (1,2,3,4,5)
WW	: Week of year(1 to 53)

TIME FORMATS

Default Time format is (HH:MI:SS)

HH	: Hour of a day
MI	: Minutes
SS	: Seconds

3) TO_DATE( )

Syntax: To_Date(Char | Num[,FMT])

This function converts from character or numeric expression to date expression.

Ex: SELECT TO_DATE('10-AUG-1990','YEAR,MONTH,DD') JD
FROM DUAL;

SPECIAL DATE FORMATES

TH : ORDINAL NUMBER
SP : SPELLED OUT NUMBER
SPTH: SPELLED OUT ORDINAL NUMBER

GROUP FUNCTIONS OR AGGREGATE FUNCTIONS

These functions act on a set of values are called as group functions. These functions produce a single value for an entire group or table.

1) MAX( ) 2) MIN( ) 3) COUNT( ) 4) SUM( ) 5) AVG( )

1) MAX( )

Syntax: MAX (Column)

This function determines the largest of all selected values of a column.

2) MIN( )

Syntax: MIN (Column)

This function determines the smallest of all selected values of a Column.

3) SUM( )

Syntax: SUM (Column)

This function determines the sum of selected Columns.

4) COUNT( )

Syntax: COUNT (* | Column)

This function determines the number of rows or not null Column values.

5) AVG( )

Syntax: AVG (Column)

This function determines the average of all selected values of Column.

Group By Clause

This clause is used with SELECT to combine a group of rows based on the values of a particular Column (or) expression.

Ex: EMP_DEPT

ENO	NAME	JOB	DNO	SAL
100	VASU	M	10	5000
101	HARI	C	20	2000
102	BOSU	C	10	1500
103	RANI	S	10	3500
104	VASU	M	30	5000
105	SRINU	C	30	2000
106	SIVA	S	20	5000
107	RAJU	C	20	2000

Ex: SELECT MAX(SAL) MAX_VAL, MIN(SAL) MIN_VAL FROM EMP_DEPT;

Ex: SELECT SUM(SAL) TOTAL_SAL ,COUNT(SAL) NO_SAL, AVG(SAL) AVG_SAL FROM EMP_DEPT;

Ex: SELECT DEPTNO,SUM(SAL) TOT_SAL FROM EMP_DEPT GROUP BY DEPTNO;

HAVING CLAUSE

This clause is applied for group of rows. This clause is used only with the group by clause.

Ex: SELECT DEPTNO,SUM(SAL) TOT_SAL FROM EMP_DEPT GROUP BY DEPTNO HAVING SUM(SAL)>=12000;

ORDER BY CLAUSE

This clause is used to display the data either ascending or descending order based on the specified table column.

Ex: SELECT * FROM EMP_DEPT ORDER BY DEPTNO;

***Insert (Query Insert) Type: SQL CATEGORY : DMLC**

Syntax: Insert into <table_name> [(COLUMN,...)] QUERY;

This command is used to copying the selected rows and columns from source table to destination table.

Ex: INSERT INTO ST SELECT * FROM STUDENT;
SELECT * FROM ST;
SELECT * FROM STUDENT;

VIEWS

After a table is created and populated with data, it may become necessary to prevent all users from accessing all columns of a table, for data security reasons. The query is stored in view permanently.

View is a virtual table or logical window. An interesting fact about a view that it stored only as a definition in ORACLE System catalogue. When a reference

is made to a view, its definition is scanned, the base table is opened and the view created on top of the base table [view does not occupies memory space].

Some views are used only for looking at table data. Other views can be used to INSSERT, UPDATE, and DELETE table data as well as view data.

TYPES OF VIEWS

1. UPDATABLE VIEW
2. READONLY VIEW

1) UPDATABLE VIEWS: A view that is used to look at table data as well as INSERT, DELETE, UPDATE table data.

2) READONLY VIEW: A view is used to only look at table data.

Features of Views

1. Security 2. Hiding Complexity 3. Renaming Columns

Syntax: CREATE [OR REPLACE] [FORCE] VIEW <VIEW_NAME>
[(COLUMN_ALIAS,...] AS QUERY
[WITH CHECK OPTION];

This command is used to create a new view.

FORCE: This clause is used to forcedly create a view without having any base table.

Query : Query is a valid Select Statement.

WITH CHECK OPTION: This option is used to prevent invalid data entry into view.

UPDATABLE VIEWS

Ex: CREATE OR REPLACE VIEW EMP_VIEW AS SELECT * FROM EMP_DEPT;

READONLY VIEWS

Ex: CREATE OR REPLACE VIEW EMP_VIEW
AS SELECT DEPTNO,SUM(SAL) TOT_SAL FROM EMP_DEPT
GROUP BY DEPTNO;

SECURITY MANAGEMENT COMMANDS

ORACLE provides extensive security features in order to safe guard information stored in its tables from unauthorized viewing and damaged. Depending on a users status and responsibility, appropriate rights on ORACLE resources (Privileges) can be assigned to the user.

ORACLE supports two types of privileges

1. System Privileges
2. Object Privileges

System Privileges

CREATE USER

Type: SQL

Syntax: Create User <Username> IDENTIFIED BY <Password>;

This command is used to create a new user_id with the password.

Note: Before using this command the user must have DBA privileges.

CONNECT

Type: SQL * Plus

Syntax: CONNECT USERNAME /PASSWORD @ HOST-STRING;

This command is used to connect from one (current) user to existed user.

SHOW USER

Type: SQL * Plus

Syntax: SHOW USER[;]

This command displays the current user name.

GRANT

CATEGORY: DCLC TYPE: SQL

Syntax1: GRANT CONNECT [,RESOURCE][,DBA] TO USER_NAME [IDENTIFIED BY <PWD>[,<PWD,- - -]];

This form of GRANT command is used to create new user_id's and assigns passwords and grants user privileges.

CONNECT: Grants a user authority to logon the specified user.

RESOURCE: Grants a user authority to create new database objects.

DBA: Grants a user authority to execute DBA commands (Data Base Administrator).

USER NAME: Is the name of valid user.

PWD: (Password) becomes the password that the corresponding user must enter to the specified ORACLE user.

Ex: GRANT CONNECT TO ABC IDENTIFIED BY CBA;
Ex: GRANT RESOURCE TO ABC;

REVOKE:

Syntax: REVOKE CONNECT[,RESOURCE][,DBA] FROM <USER>[,<USER>,..];

This command is used to remove privileges from the particular user or users.

Ex: REVOKE RESOURCE FROM ABC;
Ex: REVOKE CONNECT FROM ABC;

DROP USER **TYPE: SQL**

Syntax: DROP USER <USER_NAME> [CASCADE];

This command is used to drop the empty user.

CASCADE: This clause is used to remove the all database objects from the specified user.

OBJECT PREVILAGES

GRANT **TYPE: SQL**

Syntax2: GRANT SELECT [,INSERT] [,UPDATE] [,DELETE][,ALTER] [,INDEX] | ALL ON <TABLE_NAME> | <VIEW_NAME> TO <USER1> [<USER2>,. . .];

This form of GRANT command is used to GRANT the object privileges to the specified user or users ON particular database table or view.

Note: Before the user must have DBA privileges.

Ex: GRANT SELECT,INSERT ON EMP_DEPT TO ABC;
GRANT ALL ON EMP_VIEW TO ABC;
GRANT SELECT,INSERT,UPDATE ON EMP_DEPT TO ABC,KISHORE;

TYPE: SQL

REVOKE

Syntax: REVOKE Select [,INSERT][,UPDATE]
[,DELETE][,ALTER][,INDEX] | ALL
ON <TABLE | VIEW> FROM <USER1>
[,<USER2>,____];

This form of revoke command is used to remove the object privileges from the specified user or users on particular database table or view.

Note: User must have DBA privileges.

Ex: REVOKE INSERT,SELECT ON EMP_VIEW FROM ABC;
REVOKE ALL ON EMP_VIEW FROM KISHORE;

CONSTRAINTS

An integrity constraint is a mechanism used by ORACLE to prevent invalid data entry into the table. It's nothing but enforcing rule for the columns in a table.

ORACLE allows integrity constraints to be defined for tables and columns to enforce certain rules either within a table or between tables.

Constraints are used:

1) To enforce rules at table level whenever a row is inserted, updated or deleted from that table. The constraint must be satisfied for the operation to succeed.
2) To prevent the deletion of a table if there are dependencies from other tables.

Constraints are classified as either table constraints or column constraints.

COLUMN CONSTRAINTS

These constraints reference a single column and are defined within the specification for the column.

Syntax: [,][CONSTRAINT <CONST_NAME] NOT NULL |
CHECK (CONDITION) | UNIQUE[(COLUMN)] |
PRIMARY KEY[(COLUMN)]

TABLE CONSTRAINTS

These constraints reference one or more columns.

Syntax: ,[CONSTRAINT <CONST_NAME] UNIQUE (column,...) | PRIMARYKEY (column,...)

Various Types of Integrity Constraints:

1. Domain Integrity Constraints
2. Entity integrity Constraints
3. Referential Integrity constraints

DOMAIN INTEGRITY CONSTRAINTS

These constraints maintain value according to the specifications. NOT NULL or CHECK constraints are fall under this category.

NOT NULL: When a column is defined as not null, then that column becomes a mandatory column. It means that a value must be entered into the column.

Note: This constraint defined at the column level.

Ex: CREATE TABLE TEST(ENO NUMBER(3) NOT NULL,
NAME CHAR(15));

CHECK: This constraint must be specified as logical expression that evaluates either TRUE or FALSE.

Note: This constraint defined at the column level.

Ex: CREATE TABLE TEST
(
ENO NUMBER(3) CHECK (ENO>100),
NAME CHAR(15)
);

2) Entity Integrity Constraints

These constraints maintain uniqueness in a record, unique or primary key are fall under this category.

UNIQUE: This constraint is used to prevent the duplicate values within the rows of a specified column or columns in a table. This constraint is defined to more than one column is said to be a "Composite unique key". The maximum combination of columns that composite key can contain 32 columns. This constraint allows to insert null values also.

Ex: COLUMN LEVEL
CREATE TABLE TEST(ENO NUMBER(3) UNIQUE,
NAME VARCHAR2(10));

Ex: TABLE LEVEL
CREATE TABLE TEST(ENO NUMBER(3),NAME VARCHAR(10),
UNIQUE(ENO,NAME));

Primary Key:

This constraint avoids duplication of rows and null values. If a primary key constraint is assigned to more than one column or combination of columns, it is said to be composite primary key. Which can contain maximum of 32 columns.

Note: A table may have only one primary key. (Either column or table)

A single column primary key is called a SIMPLE KEY.
A multicolumn primary key is called a composite primary key.

Ex: column level
CREATE TABLE TEST(ENO NUMBER(3) PRIMARY KEY,
NAME CHAR(10));

Ex: table level
CREATE TABLE TEST(ENO NUMBER(3),NAME CHAR(10),
PRIMARY KEY(ENO,NAME));

DISPLAY THE ALL CONSTRAINT NAMES AND CONSTRAINT TYPES OF A PARTICULAR TABLE

1) SELECT CONSTRAINT_NAME FROM USER_CONSTRAINTS
WHERE TABLE_NAME='TEST';

2) SELECT CONSTRAINT_TYPE FROM USER_CONSTRAINTS
WHERE TABLE_NAME='TEST';

3) SELECT CONSTRAINT_NAME,CONSTRAINT_TYPE,
SEARCH_CONDITION FROM USER_CONSTRAINTS
WHERE TABLE_NAME='TEST';

Multiple Constraints in a Single Column

Column_name type not null [check(condition)] [unique] [primary key] [references master_table (unique | primary key column)]

Ex: CREATE TABLE EMP_TEST
(ENO NUMBER(4) NOT NULL CHECK(ENO>1000),
NAME CHAR(15));

Ex: CREATE TABLE EMP_TEST1
(ENO NUMBER(3)
CHECK(ENO BETWEEN 100 AND 200) UNIQUE ,
NAME CHAR(15));

ALTER TABLE(CONSTRAINTS) Type: SQL

This command is used to add, modify, disable, enable, or remove the CONSTRAINTS from the table.

Syntax: Alter table <table_name> add [constraint <const_name>]
UNIQUE(column,...) | CHECK (column_condition) |
PRIMARY KEY (column,...);

ADD: This key word is used to add a new constraint to a specified column or columns.

Ex: Create table test(eno number(3),name char(10));
Alter table test add constraint ch_eno check(eno between 1and 100);

MODIFY: This key word is used to add a NOT NULL constraint t a specified constraint.

Ex: create table test(eno number(3),name char(10));
Alter table test modify eno number(3) constraint n_en not null;

DISABLE: This keyword is used to delete an existed constraint temporarily.

Ex: alter table test disable constraint n_en;

ENABLE: This keyword is used to enable the previously disabled constraint.

Ex: alter table test enable constraint n_en;

DROP: This keyword is used to delete an existed constraint (permanently)

Ex: Alter table test drop constraint n_en;

REFERENTIAL INTIGRITY CONSTRAINTS

These constraints enforce relationship between the tables. FOREIGN KEY constraint is fall under this category.

FOREIGN KEY

This constraint is used to establish master_detail or parent_child relationship between two tables having a common column, we may use of referential integrity constraints. To implement this we should define the column in the master table as a unique or primary key and the same column in the detailed table as a foreign key referencing to the corresponding parent entry. It allows null values.

REFERENCE KEY

A unique or primary key, which is defined on the column, belongs to the master table.

CHILD TABLE OR DETAILED TABLE

This table depends upon the values present in the referenced key of the master table, which is referred by a foreign key.

Master table / Parent table

This table determines whether insertion or updatation of data can be done in child table. This table would be referenced by child table foreign key.

Note: 1. Master or Parent tables are independent tables.
2. Child or detailed tables are dependent tables.

Column Constraints

Syntax:

[Constraint <const_name>] REFERENCES
Master_table (unique| primary key column)
[On Delete Cascade]

Ex: MASTER AND DETAIL TABLES

```
CREATE TABLE MAST
(ENO NUMBER(3) PRIMARY KEY, NAME CHAR(10));

CREATE TABLE CHILD
(ENUM NUMBER(3) REFERENCES MAST(ENO),
SAL NUMBER(9));
```

```
INSERT INTO MAST VALUES(1,'RAJU');
INSERT INTO CHILD VALUES(1,2500);

DELETE FROM MAST;
ERROR.....
```

Table Constraints

Syntax: ,[Constraint <const_name>] FOREIGN KEY (child table FOREIGN KEY column. . .) REFERENCES Master Table(Primary unique column,. . .) [On delete Cascade]

Ex: MASTER AND DETAIL TABLES

```
CREATE TABLE MAST
(ENO NUMBER(3),NAME CHAR(10),
PRIMARY KEY(ENO,NAME));

CREATE TABLE CHILD
(ENUM NUMBER(3),NAME CHAR(10),SAL NUMBER(9),
CONSTRAINT FK_EN_NA FOREIGN KEY(ENO,NAME)
REFERENCES MAST(ENO,NAME));
```

✓ **On Delete Cascade:** ORACLE provides an option for a Foreign Key constraint that causes the deletion of a Primary Key value cascade to any child records reference that value.

Ex:

```
CREATE TABLE TEST(ENO NUMBER(3) UNIQUE,
NAME CHAR(10));

CREATE TABLE TEST1(ENUM NUMBER(3)
REFERENCES TEST(ENO) ON DELETE CASCADE,
SAL NUMBER(4));

INSERT INTO TEST VALUES(1,'RAJU');
INSERT INTO TEST1 VALUES(1,2000);

DELETE FROM TEST WHERE ENO=1;
```

Default

By using the default clause, when defining a column we can establish a default value for that column. This default value is used for a column whenever a row is inserted into the table without specifying that column in the insert statement.

Syntax: Column_Name specification DEFAULT_VALUE

Ex: CREATE TABLE EMP_T
(ENO NUMBER(3),JOB CHAR DEFAULT 'C');

INSERT INTO EMP_T VALUES(10,'M');
INSERT INTO EMP_T(ENO) VALUES(11);
SELECT * FROM EMP_T;

DROP TABLE

Syntax:

Drop table <Table_Name> CASCADE CONSTRAINTS;

Cascade: This clause is used to break the relations (delete the constraints) between Master and Detailed table. This command is used to delete a Master table.

Ex: DROP TABLE TEST;
DROP TABLE TEST CASCADE CONSTRAINTS;

JOININGS & UNIONS

Joins and Unions are two techniques used to group together data stored in different tables. These two methods of grouping bring data together from multiple tables that are related to each other.

The join operation is the mechanism that allows tables to be related to one another. The join operation retrieves columns from two or more tables.

Main Points:

1. Joins are used to combine columns from different tables.
2. In a join the tables are listed in the from clause, separated by commas.
3. The condition of the query can refer to any column of any table join.
4. The connection between tables is established through the where clause.

Types of Joins

1. Inner Joins **2. Cartesian Join** **3. Outer Join** **4. Self Join**

1. Inner joins

a) Equi Join

When two tables are joined together using equality of values in one or more columns they make an equi join. The equi join operator is =.

Note: Ambiguous Columns

You need to keep in mind that each reference to a column in a join must be unambiguous. In this context unambiguous means that if the column exists in more than one table.

b) Non-Equi Join

The relationship is obtained using an operator other than equal (=) sign.

2) Outer Joins

If there are many values in one table that do not have corresponding values in the other table, in an equi join that row will not be selected. Such rows can be forcefully selected by using the Outer Join symbol (+). The corresponding Columns for that row will have NULLS.

3) Self Join

To join a table to itself means that each row of the table is combined with itself and with every other row of the table. The Self Join can be viewed as a join of two copies of the same table. The table is not actually copied.

4) Cartesian Joins

You are first learning to join multiple tables, a common error is to forget to provide a join condition in the where clause. If you forget a join condition you will notice two things.

1. The query takes longer to execute.
2. The query-retrieved records are much longer than you expected.

When nowhere clause is specified, each row of one table matches every row of the other table. This result is a Cartesian product.

Set Operators

The set operators combine two or more queries into one result.

1) Union **2) Union all** **3) Interset** **4) Minus**

Note: 1) Multiple queries can be merged together and their results combined.
2) The queries are all executed independently but their output is merged.
3) Only the final query ends with a semicolon.
4) The number of columns in the queries must be the same and of the same data type.

Union

This operator is used to combines the results, which eliminates duplicated selected rows.

Union All

This operator combines the results which does not eliminate duplicated selected rows.

Intersect

This operator combines the results, which returns only those rows returned by both queries.

Minus

This operator combines the results, which returns only those rows returned by the first query but not in the second query.

Ex: select desig from emp where deptno=10
union
select desig from emp where deptno=20;

select desig from emp where deptno=10
union
select desig from emp where deptno=20
union
select desig from emp where deptno=30;

SUBQUERIES

A query within another query is known as sub query.

1. The result of one query is dynamically substituted in the condition of another query.
2. SQL first evaluates the sub query (or Inner Query) within the where clause.
3. The return value of sub query is then substituted in the condition of the outer Query.

4. There is no limitation to the level of nesting queries.
5. When using relational operators, ensure that the sub query returns a single row output.

Syntax: Select Column_List From Table Where Column=(Select Columns from Table where- - -);

Note: The Sub query always must be with in the parenthesis.

Sub query Operators

1) IN 2) ANY 3) ALL

1) **In**

Syntax: Column_Name[not] IN (value1,value2,. . .)

This operator defines set of values in which a value may be existed or not.

Ex: select * from emp
where sal in(select sal from emp where job in('c','C'));

2) **Any or some**

Syntax: Column_Name ROP ANY

ROP: Relational Operator(>,>=,<,<=)

This operator compares a value to each value in a list.

Ex: select * from emp
where sal>=any(select sal from emp where job in('c','C'));

>=ANY

Greater than or equal to minimum salaries returned by a sub query salary.

<ANY

In this example it displays all rows from Emp where the salary is less than max salary returned by a sub query salaries.

<=ANY

Less than or equal to max salary returned by a sub query salaries.

=ANY

It is nothing but IN operator.

3) **ALL**

Syntax: column_Name ROP ALL

This operator compares a value to every value in list.

Ex: select * from emp
where sal>=all(select sal from emp where job in('c','C'));

>ALL

In this example it retrieves all rows from Emp table where the Sal is greater than Max Salary returned by a sub query salaries.

>=ALL

Where the salary is >= Max Sal returned by a sub query salary.

<ALL

In this example it list out all records from Emp table where the salary is less than minimum salary returned by a sub query salary.

<=ALL

Where the salary is <= Min salary returned by a sub query salary.

CORRELATED SUB-QUERIES

Correlated sub query is evaluated on once per a row processed by the parent statement. It is signaled by a column name, a table name or table alias in where clause. It is used to answer Multipart questions whose result depends on the value of each row of parent query.

Ex: select * from emp a
where 3>(select count(*) from emp where sal>a.sal);

ROWID

RowId is a row identifier. Every row has the unique ID that corresponds to the Block number, File Number, and relative row number.

RowNum

RowNum is a pseudo column that indicates order of the retrieved row.

Ex: select eno,name,sal,rowid from emp;
Ex: select eno,name,sal,rownum from emp;

Transactional Control Language Commands (TCLC)

1) Commit **2) Rollback** **3) SavePoint**

COMMIT

TYPE: SQL

Syntax: COMMIT;
This command is used to save the **all** DMLC transactions.

ROLLBACK

Syntax: ROLLBACK [To save point <save point_name>];
This command is used to discards all DMLC operations.

Ex:
```
INSERT INTO TEST VALUES(1,'VASU');
COMMIT;

INSERT INTO TEST VALUES(2,'RAJU');

ROLLBACK;

SELECT * FROM TEST;

DELETE FROM TEST;

SELECT * FROM TEST;
```

SAVEPOINT

Syntax: savepoint <savepoint_name>;

Ex:
```
SAVEPOINT FIRST;

INSERT INTO TEST VALUES(11,'RAMESH');

SAVEPOINT SECOND;

UPDATE TEST SET NAME='KISHORE';

SAVEPOINT THIRD;

DELETE FROM TEST;
```

SELECT * FROM TEST;

ROLLBACK TO SAVEPOINT THIRD;

SELECT * FROM TEST;

TRUNCATE TABLE **Type: SQL**

Syntax: TRUNCATE Table <Table_Name>;

This command is used to delete all rows permanently from the specified database table.

Ex: TRUNCATE TABLE TEST;

INDEXES

When the user fires 'select' statement to search for a particular record, the Oracle engine first locate the table name on the hard disk. The oracle engine then performs a sequential search to locate records that match user given condition value.

Indexing a table is an Access Strategy that is a way to sort and search records in the table. Indexes are essential to improve the speed with which records can be located and retrieved from a table.

Oracle allows the creation of two types of indexes.

1. DUPLICATE INDEX **2. UNIQUE INDEX**

1. DUPLICATE INDEX

This index allows duplicate values for the indexed columns.

An index can be created on one or more columns, based on the number of columns the index can be classified into two.

Simple Index

An index created on a single column of a table.

Syntax: Create Index <Index_Name> ON Table_Name (Column);

Ex: create index emp_ind on emp(eno);

Composite index

An index created on more than one column of a table.

Syntax: create index <index_name> on table_name(column,...);

Ex: create index emp_ind on emp(eno,name);

2) UNIQUE INDEX

This index prevent duplicate values for the indexed columns.

Simple unique Index

An index created on a single column of a table.

Syntax: Create unique Index <Index_Name> ON Table_Name (column);

Ex: create UNIQUE index emp_ind on emp(eno);

Composite unique index

An index created on more than one column of a table.

Syntax: create index UNIQUE < index_name> on table_name (column...);

Ex: create unique index emp_ind on emp(eno,name);

Note: After creating Unique Index it prevents duplicate values into the specified table column.

CLUSTERS

Clusters are used to save the disk space and improve the performance of the database.

CREATE CLUSTER **TYPE: SQL**

Syntax: Create CLUSTER <Cluster_Name> (Cluster Column data type,...);

This command is used to create the new Cluster.

Ex: CREATE CLUSTER EMP_CLU
(TENO NUMBER(4));

CREATE TABLE **TYPE: SQL**

Syntax: Create table Table_Name (Column data type,...) Cluster <Cluster_Name> (Table_Column,...);

This form of create table command is used to create a new Cluster Table.

Note: 1) The CLUSTER column specification (data type width) and CLUSTER table specification must be the same.
2) Before inserting rows into CLUSTER table the specified CLUSTER must be indexed.

Ex: CREATE TABLE EMP
(ENO NUMBER(4),NAME CHAR(15))
CLUSTER EMP_CLU(ENO);

CREATE INDEX (CLUSTER INDEX) TYPE: SQL

Syntax: CREATE INDEX <INDEX_NAME> ON CLUSTER Cluster_Name;

This command is used to create a Cluster Index.

Ex: CREATE INDEX T_CLU ON CLUSTER EMP_CLU;

SEQUENCE TYPE: SQL

Syntax: CREATE SEQUENCE <SEQ_NAME>
START WITH <M>
[INCREMENT BY <N>]
[MAX VALUE <M1>]
[MIN VALUE <M2>]
[CYCLE | NOCYCLE];

ORACLE provides an object called 'SEQUENCE' that can generate numeric values. A sequence can be defined to

1) Generate No's in ascending | descending order.
2) Provide intervals between numbers.

A SEQUENCE is an independent object and can be used with any table that requires its output.

START WITH <M>

'M' is the starting number of the SEQUENCE.

INCREMENT BY <N>

Specifies the interval between sequence No's. It can be either + ve or –ve.

MAXVALUE

Specifies the maximum value that a sequence can generate.

MINVALUE

Specifies the sequence minimum value.

CYCLE

Specifies that the sequence continuous to generate repeat values after reaching either its maximum value.

NOCYCLE

Specifies that the sequence cannot generate more values after reaching the maximum value.

NEXTVAL

It is pseudo column, which is used to generate next sequence number.

Syntax: SEQUENCE_NAME.NEXTVAL

CURRVAL

It is a pseudo column, which is used to displace or insert the current sequence number.

Syntax: SEQUENCE_NAME.CURRVAL

```
Ex: CREATE SEQUENCE ABC START WITH 100;
    INSERT INTO EMP VALUES(ABC.NEXTVAL,'VASU');
    SELECT * FROM EMP;
    SELECT ABC.CURRVAL FROM DUAL;
```

SYNONYM **TYPE: SQL**

SYNTAX: CREATE [PUBLIC] SYNONYM <SYNONYM_NAME>
FOR [USER_NAME.] TABLE_NAME;

SYNONYM is a mirror image of the particular database table.

ORACLE supports two types of SYNONYMS.

1. PRIVATE 2. PUBLIC

Private synonyms are created by the user.

Public synonyms are created by the DBA.

Ex: CREATE SYNONYM EMP_DUP
FOR EMP;
SELECT * FROM EMP;
SELECT * FROM EMP_DUP;

ENVIRONMENT COMMANDS OR SET COMMANDS (SQL * PLUS)

These commands effects only on current SQL * Plus session. These commands are used to set the current Sql * Plus environment. It means either screen settings or data settings.

Note: Frequently used settings may be placed in the file called *login.Sql*.

1. SET AUTOCOMMIT
2. SET HEADING
3. SET NUMWIDTH
4. SET FEEDBACK
5. SET LINESIZE
6. SET PAGESIZE
7. LONG
8. SET TIME
9. SET TIMING
10. SET PAUSE
11. SET SQLPROMPT
12. SET SQLPROMPT
13. SET VERIFY
14. SET TERMINATOR
15. SET NULL
16. SET SQLCASE
17. SET DEFINE
18. SET SCAN
19. SET SQLNUMBER
20. SET HEAD SEPARATOR
21. SET NEWPAGE
22. SET SERVEROUPUT

SPECIAL SQL*PLUS COMMANDS

1) SPOOL **2) PASSWORD**

SET AUTOCOMMIT

Syntax: SET AUTOC[OMMIT] ON | OFF

This command controls commits transactions. It is set to ON, SQL*PLUS performs a commit after each SQL statement (DMll) is processed. The default setting for auto commit is OFF.

SET HEADING

Syntax: SET HEAD[ING] ON | OFF

By default SQL * PLUS displays column headings. To suppress the display of column headings, simply set heading to off.

SET FEEDBACK

Syntax: SET FEEDBACK ON | OFF

Syntax: SET FEEDBACK <N>

This SETFEEDBACK<N> command controls when and whether SQL*PLUS indicates the Number of rows returned by a query. The default settings of FEEDBACK are 6(records). In other words, if a Query retrieves 6 or more rows, SQL * PLUS displays the no. of rows. If the query retrieves < 6 rows, SQL * PLUS will not provide this feedback.

If set feedback is set to off, Sql * Plus suppresses the displaying records message.

SET NUMWIDTH

Syntax: SET NUMWIDTH <N>

SQL*PLUS uses the value of NUMWIDTH to determine the width to use when displaying numbers.

The default value of NUMWIDTH is 10. If we want to increase or decrease this value, if your application requires it.

SET LINESIZE

Syntax: SET LINESIZE <N>

This command controls the maximum number of characters that appear on the line of output.

If you want to increase and decrease the line size RESET line size followed by number N.

SET PAGESIZE

Syntax: SET PAGESIZE <N>

This command defines the number of lines on a page, which determines when column headings and Page titles should be displayed. If you want to suppress all page titles and column headings.

SET TIME

Syntax: SET TIME ON | OFF

We can include the current time with SQL*PLUS prompt. You may find this setting useful when spooling output to a file.

SET TIMING

Syntax: SET TIMING ON | OFF

This command sets the display of timing statistics for each SQL Command

To on or off. If you are trying to collect performance information about ORACLE8, you can set timing to on.

SET SPACE

Syntax: SET SPACE <N>

This command is used to sets the number of places between columns in a table display. The default is / and the maximum value of N is 10.

SET SQLPROMPT

Syntax: SET SQLP[ROMPT] 'TEXT'

This command is used to change the current SQL prompt.

SET SQLTERMINATOR

Syntax: SET SQLT[ERMINATOR] 'CHAR'

This command is used to change SQL statement termination character (;)

SET PAUSE

Syntax: SET PAUSE ON|OFF (OR) SET PAUSE 'TXT'

If you submit a query that retrieves many records, the default behavior of SQL * PLUS is to Send the results zipping by you on the screen. If you set pause to on to a string, SQL*PLUS waits for you to press the enter key before continuing the next screen's output.

SET ECHO

Syntax: SET ECHO ON|OFF

If you don't want to echo the SQL statements that SQL * PLUS set echo to off. OFF is commonly used to prepare reports in which you want to see only the desired results without the SQL statements.

SET VERIFY

Syntax: SET VERIFY ON | OFF

SET VERIFY ON makes SQL * PLUS displays the test of command line before, after replacing a substitution variables reference with the values value. Off suppress the display.

SET SQL CASE

Syntax: SET SQLCASE MIXED | UPPER |LOWER

This command is used to sets the data case, whenever the data is inserted or updated label columns. The default data case is MIXED.

SET UNDERLINE

Syntax: SET UNDERLINE CHAR

This command is used to set the underline the given string character.

SET NULL

Syntax: SET NULL CHAR

This command is used to sets the text that SQL * Plus to represent a null value.

SET DEFINE

Syntax: SET DEFINE 'CHAR' AMPERSAND

Normally, the & denotes a variable. This can be changed with set definition followed by single symbol.

SET SCAN

Syntax: SET SCAN ON | OFF

This command turns variable substitution ON or OFF. If SETSCAN is OFF, any variable in a select statement is treated as a literal.

SET SQL NUMBER

Syntax: SET SQL NUMBER ON | OFF

This command is used to display the line number or hide the line number.

SET HEADSEPARATOR

Syntax: SET HEADS[EPARATOR] 'CHAR'

This heading separator identifies the single character that tells SQL * Plus to split a title on two or more lines.

SET LONG

Syntax: SET LONG <N>

Generally SQL*Plus will not display more than 80 characters of a long column, unless you increase the long variable use set long < No. of chars>

SQL * PLUS REPORT WRITER COMMANDS

SQL*PLUS is usually thought of as a kind of interactive report writer. It uses SQL to set information from ORACLE database and lets you create well-formatted reports by giving you. Easy control over titles, column headings, sub totals, and totals, reformatting of numbers and text, and much more.

FORMATTING OUTPUT

These commands enable you to specify formats, tables and sub totals to suppress repeating values. The formatting commands are temporarily. They remain in effect only for SQL statements processed during the same SQL sessions.

FORMATTING A COLUMN WITH THE COLUMN COMMAND

The column command offers many options formatting a columns display. This command can over ride default output formatting that SQL*Plus provides.

SPECIFYING A COLUMNS FORMAT

Each column depending on its data type as a format that can be specified for displaying output.

FOR NUMERIC COLUMNS

You can use format option to specify the following format signs

(9 $ PR . , mi)

Syntax: COLUMN <COLUMN_NAME> FORMAT <FMT>

Format option to specify the displayed width of a character column.

Syntax: COLUMN<COLUMN_NAME> FORMAT A<N>
[WORD_WRAP | TRUNK]

TRUNC – In one line N class are displayed remaining are skipped.
WORD – WRAP This option is used to prevent mid word wrapping.
TRUNC This option is used to truncate the display of a column contents.

CHANGING A COLUMNS HEADING

When you specify a list of columns to be displayed in a query, SQL * Plus uses the column name as the column heading. If you want to use a different column heading, you have two choices.

1. Either uses a column alias in select statement.
2. Either uses the column command to specify a different column heading.

Note: If you want to use more than one word as the column heading, enclose the column heading in double quotes.

Syntax: COLUMN<COLUMN_NAME> HEADING 'TEXT'

SPLITTING THE COLUMN HEADING

SQL * PLUS supports a special character (|) that is used to splitting the columns reading.

DISTINCT COLUMN DISPLAY ATTRIBUTE

This command is used to show the column display attributes.

Syntax: COLUMN <COLUMN_NAME>

LISTOUT THE ALL COLUMN DISPLAY ATTRIBUTES

Syntax: COL / COLUMN

DISPLAY AND HIDE COLUMN DISPLAY ATTRIBUTES

Syntax: COLUMN <COLUMN_NAME> ON
COLUMN < COLUMN_NAME> OFF

CLEAR THE SPECIFIED COLUMN ATTRIBUTES

Syntax: COLUMN <COLUMN_NAME> CLEAR

CLEAR ALL COLUMN DISPLAY ATTRIBUTES

Syntax: CLEAR COLUMNS

SUPPRESSING DUPLICATE VALUES WITH BREAK COMMAND

1. BREAK: This command instructs SQL * Plus to suppress the display of repeating values for a column.

Syntax: BREAK ON COLUMN | REPORT [SKIP <N>]

2. COMPUTE: **COMPUTE SUM | AVG | MAX | MIN | COUNT OF <COLUMN> BREAK ONSKIP<N>]**

Oracle provides a built in function for computing a total sum function. But SQL * PLUS also provides the compute command for calculating subtotals, averages, and totals.

Note: You can compute more than one function using the compute command.

TO DISPLAY THE INFORMATION ABOUT ALL COMPUTE COLUMNS

Syntax: COMPUTE

TO DISPLAY THE INFO ABOUT ALL BREAK COLUMNS

Syntax: BREAK

CLEAR BREAKS

This command clears all break columns that are previously defined.

Syntax: CLEAR BREAKS

CLEAR COMPUTES

This command clears all compute columns that are previously defined.

Syntax: CLEAR COMPUTES

ADDING TITLES TO SQL * PLUS OUTPUT

SQL * PLUS provides two commands for producing titles.

1. TTITLE 2. BTITLE

TTITLE (TOP TITLE)

Syntax: TTI[TLE] CENTER | LEFT | RIGHT 'TEXT'
CENTER | LEFT | RIGHT 'TEXT'- - -[SKIP N]

This command is used to define the page top title.

BTITLE (BOTTOM TITLE)

Syntax: BTI[TLE] CENTER | LEFT | RIGHT | 'TEXT'
CENTER | LEFT | RIGHT TEXT [SKIP<N>]

This command is used to define bottom titles of a page.

PASSWORD

This command will ask you to change the password for the currently logged in user. It can also be used to change the password of another user if logged in as DBA.

Syntax: PASSWORD

SPOOL

You can use either of two methods to save sql statements to a file.

1. You can use the save command to save only the current contents of the buffer to a file.

2. You use spool command, it saves all output from sql * plus – Each SQL statement and its output to a file. The default extension of a spool file is .LST

CODD RULES

Dr. EF. CODD formulated 12 rules for RDBMS in 1970. In addition to the 12 rules, there exists a rule called RULE.

A relational system must be able to manage databases entirely throughout its relational capabilities.

RULES

1. Information Representation
2. Guaranteed Access
3. Systematic Treatment of null values
4. Database description
5. Comprehensive data sub language (DDLC)
6. View updating
7. High level update, insert, delete (DMLC)
8. Physical data Independence
9. Logical data Independence
10. Distribution
11. Non-Subversion
12. Integrity

1) INFORMATION REPRESENTATION

In RDBMS all information is explicitly and logically represented by data values in tables.

Ex: If you create a new database object (table, view, synonym etc.,) by giving SQL command. If the given object name did exist system prompts that the table or view already exist.

In this case table name already exist were possible, because the information is stored in the system catalogues commonly known as data dictionaries. The information such as table and column names is contained in the form of tables.

2) GUARANTEED ACCESS

It is fact the table can be taken as a storage structure, the intersection of each column and row there will necessarily only one value of data (or null).

Every item of data logically addressed by a combination of table name and column name.

3) SYSTEMATIC TREATMENT OF NULL VALUES

In RDBMS NULL values are supported for the representation of missing and inapplicable information.

This rule states that support NULL values must be consistent throughout the RDBMS and independence of data types of the column.

A NULL value in an integer column, a NULL value in a character column must mean the same as a NULL in date column.

4) DATABASE DESCRIPTION RULE

A description of the database is stored and maintained in the form of tables. This allows the users with appropriate authority to query such information in the same way and using the same language as they would any other data in the database.

5) COMPREHENSIVE DATA SUB LANGUAGE (DDLC)

RDBMS must be manageable through its own extension of SQL.

This language should support data definitions, views, integrity constraints, authorization, and transaction boundaries.

6) VIEW UPDATING RULE

Views that can be updated in theory, can also be updated by the system itself. Though it is possible to create views in all sorts of logical ways and with aggregates and virtual columns (temporarily). But it is not possible to update, insert, delete.

7) HIGH LEVEL UPDATE, DELETE, INSERT

RDBMS has to be capable of high level inserting, updating and deleting data.

8) PHYSICAL DATA INDEPENDENCE

The user access to the database via monitors or application programs, must remain logically consistent whenever changes to the storage representation.

9) LOGICAL DATA INDEPENDENCE

A single table should be divisible into one or more other tables, provided it preserves all the original data (non loss) and maintains the primary key in each and every fragment or table.

10) DISTRIBUTION RULE

This is one of the most attractive aspects of the RDBMS. Database systems built on the relational framework are well suited to today's C/S data base design (Client / Server).

11. NON – SUBVERSION RULE

RDBMS supports a lower level language that permits for example, row at a time processing, and then this language must not be able to bypass any integrity rules or constraints defined in the high level.

12) INTEGRITY RULE

All integrity constraints defined for a database must be definable in the language, and stored in the database as data in tables.

RULE 0

Any truly relational database must be manageable entirely through its own relational capabilities.

NORMALIZATION

Normalization is the process of putting things right. The origin of this term is the Latin word 'Norma'. In RDBMS the term 'Normal' also has a specific mathematical meaning having to do with separating elements of data into groups and defining the normal or right relationships.

Normalization theory is the study of relations (tables), attributes (columns) and the dependency of attributes upon one another.

GOALS OF NORMALIZATION

Minimizing redundant data (duplicate date).
Minimizing inconsistent data.
Designing Data Structures for easier maintenance.

UNNORMALIZED DATA STRUCTURES

An unnormalized data structure contains redundant and disorganized which needs to be organized by dividing the data over several tables to avoid redundancy. This is achieved by going through the process of normalization.

NORMALIZATION PROCESS

Normalization is achieved in terms of Normal forms, which are mainly defined as 3 forms.

1. First Normal Form (I NF)
2. Second Normal Form (II NF)
3. Third Normal Form (III NF)

FIRST NORMAL FORM (I NF)

First normal form is very simple to achieve. The only requirement is that each cell in the table should have a single value (OR) All columns should contain a single piece of information.

Main Points:

1. Identity repeating groups of fields.
2. Remove repeating groups to a separate table.
3. Identify the keys for tables.

SECOND NORMAL FORM (II NF)

This form is achieved when an attribute of table is not functionally dependent on only one column of a concatenated key.

Main Points:

1. Check if all fields are dependent on the primary key.
2. Remove fields that depend on part of the key.
3. Dependent fields as a separate table.

THIRD NORMAL FORM (III NF)

ALL columns depend on the primary key and nothing but the primary key.

Main Points:

1. Remove fields that depend on other non-key fields.
2. Interdependent fields as separate table, identify the key and name of the table.

OBJECTS

As of ORACLE8, you can extend your relational database to include Object Oriented concepts. Infact the database is referred to as an ORDBMS (Object Relational Database Management System).

The implication for developers is that 3 different flavours of ORACLE are available.

1. RDBMS 2. ORDBMS 3. OORDBMS

RDBMS:

The traditional ORACLE RDBMS, extended to include Object Oriented Concepts and structures, such as Abstract Data Types (ADT), Varying Arrays (VA), Nested Tables (NT).

ORDBMS

RDBMSs are the standard tool for managing data. They provide fast, efficient and completely reliable access to huge amounts of data for millions of businesses around the world every day.

The objects option makes oracle an Object Relational Database Management System (ORDBMS), which means that users can define additional kinds of data specifying both the structure of the data and the ways of operating on it and use these types within the relational model. This approach adds value to the data stored in a database.

USER DEFINED DATA TYPES

The objects option adds two categories of user defined data types to oracle.

1. Object Types 2. Collection Types

1) **Object Types:** An object type is a user defined composite data type that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called attributes.

An object type with 3 kinds of components.

1. name 2. attribute 3. methods

1) **name:** which serves to identify the object type uniquely.
2) **attribute:** which model the structure and state of the real world entity. Attributes are built-in types or user defined types.
3) **methods:** which are functions or procedures written in PL/SQL and stored in the database.

An object type is a template. A structured data unit that matches the template is called an Object.

OBJECT

Combination of Data and Methods (Functions or procedures) makes up an object.

Creating Object Types:

ABSTRACT DATATYPE

Abstract data types are user created data types that consists of one or more subtypes. Rather than being constrained to the standard ORACLE data types of number, varchar2, date etc. Abstract data types can more accurately descrit e your data.

Syntax: CREATE [OR REPLACE] TYPE <TYPE_NAME> {IS | AS}
OBJECT (ATTRIBUTE_NAME DATATYPE, . . .);

This command is used to create new abstract data type.

Ex:
```
create type person_addr as object
( drno varchar2(10),
  city varchar2(15),
  state varchar2(7),
  pin number(7)
)
/
```

Points to remember

1. When defining object types you cannot define constraints.
2. You cannot insert data into object types.
3. You can use object types in object types.

DROP TYPE

Syntax: DROP TYPE <TYPE_NAME> [FORCE];

CREATING VIEWS (OBJECT VIEWS)

ORACLE provides object views as means for defining objects used by existing relational tables.

VARYING ARRAYS

Varying Arrays and nested tables are called collectors. Collectors make use of abstract data types.

You should be familiar with the creation and implementation of ADT'S, before attempting to use Varying Arrays and nested tables.

A Varying Array allows you to store repeating attributes of a record in a single row.

Syntax: CREATE [OR RELACE] TYPE <TYPE_Name> AS VARRAY (SIZE) OF DATATYPE (SIZE);

SELECTING DATA FROM VARYING ARRAYS

When you insert records into a varying array, you need to make sure that the number of entries you insert into the varying array does exceed its maximum. The maximum number of entries in the varying array is specified when the varying array is created.

You cannot, query the varying array directly via a select command. In order to retrieve the data from a varying array, you can use a set of nested cursor for loops.

PL / SQL

PL / SQL is a sophisticated programming language used to access an ORACLE database.

PL / SQL is integrated with the database server, so that the PL / SQL code can be processed quickly and efficiently.

ORACLE is a relational database. The language used to access a relational database is SQL. SQL is a flexible, efficient language, with features designed to manipulate and examine relational data.

SQL is a fourth generation language. This means that the language describes what should be done, but not how to do it.

DISADVANTAGES OF SQL

1. SQL does not have any procedural capabilities. i.e. SQL does not provide the programming techniques of conditional checking and looping that is vital for testing before storage data.

2. SQL statements are send to the ORACLE engine one at a time. Each time an SQL statement is executed, a call is made to the engine's resources. This adds to the traffic on the network, and decreasing the speed of data processing, especially in a multi-user environment.

3. While processing an SQL statement if an error occurs, the ORACLE engine displays its own error messages.

Third generation languages, such as C and COBOL are more procedural in nature. A program in a third generation language implements a step-by-step algorithm to solve the problem. These languages introduce the procedural constructs that are useful to express a desired program. This is where PL / SQL comes in – if combines the power, and flexibility of SQL with the procedural constructs of a 3rd generation language.

PL / SQL stands for procedural language for SQL.

ADVANTAGES OF PL / SQL

1. PL / SQL is a development tool that supports SQL DMLC and TCLC commands and also provides facilities of conditional checking and loop controls.

2. PL / SQL sends an entire block of statements to the ORACLE engine at a time. This reduces network traffic. The ORACLE engine gets the SQL statements as a single block.

3. PL / SQL also permits dealing with errors as required and facilitates displaying user-friendly messages, when errors are encountered.

4. PL / SQL allows declaration and use of variables in a blocks of code. These variables can be used to store intermediate results of a query for later processing.

5. Via PL / SQL all sorts of calculations can be done quickly and efficiently with the use of the ORACLE engine.

6. Applications written in PL / SQL are portable to any computer hardware and operating system, where ORACLE is operational. Hence PL / SQL code block written for a DOS version of ORACLE will run on its UNIX version, without any modifications at all.

FEATURES OF PL / SQL

1. Block structure program: The basic unit in PL / SQL is a block. All PL / SQL programs are made up of blocks which can be nested within each other. Each block performs a logical unit of work in the program. A PL / SQL block has the following structures.

[DECLARE]

This section consists of PL / SQL variables, types, cursors, user defined exceptions and local sub pages.

BEGIN

EXECUTABLE_SECTION

This section consists of procedural and SQL statements. This is the main section of the PL / SQL block.

[EXCEPTION]

This section consists of error handling statements.

END;

The declarative and exception handling sections are optional. Only the executable section is required. The executable section must also contain al least one executable statement.

1. Error Handling
2. Variables and types
3. Looping constructs
4. Cursors
5. Functions

!= Not equal operator
Not equal operator
Not equal operator
<= Less than or equal operator
>= Greater than or equal operator
** Exponentiation operator
/* Initial multiline comment indicator
*/ Final Multiline comment indicator
=> Association operator

KEYWORDS OR RESERVE WORDS

Many identifiers known as keywords have a special meaning to PL / SQL. It is illegal to use these words to name your own identifiers.

LITERALS (CONSTANTS)

A literal is a character, numeric or Boolean value that is not an identifier.

COMMENTS

Comments improve readability and make your programs more understandable. They are ignored by the PL / SQL compiler. PL / SQL supports two kinds of comments.

1. Single line comments.
2. Multiline Or C-style comments.

SINGLE LINE COMMENTS

A single line comments starts with the two dashes (–) hyphen and continuous until the end of the line.

MULTLILINE COMMENTS

Multiline comments start with the /* delimiter and end with the */.

Note: If the comment extends over more than one line the double dash is necessary at the start of each line.

PL / SQL DATA TYPES

PL / SQL introduce 3 categories of data types.

1. Scalar data types
2. Composite data types
3. Reference Data types

PL / SQL 8.0 defines an additional type category is LOB type (Large Objects).

Note: PL / SQL data types are defined in a package called 'standard'.

SCALAR DATA TYPES

The legal scalar data types consist of the same types valid for a database column with a number of additions. The scalar data types can be divided into 6 families.

1. Numeric 2. Character 3. Raw 4. Date
5. Rowid 6. Boolean

BOOLEAN

Boolean variables are used in PL / SQL control structures, such as if else and loop statements. A Boolean value can hold TRUE or FALSE.

VARIABLE DECLARATIONS

Communication with the database takes place through variables in the PL / SQL block.

Variables are memory locations, which can store data values. These variables are declared in the declarative section of the block. Every variable has a specific type as well, which describes what kind of information can be stored in it.

Syntax: VARIABLE_NAME TYPE [CONSTANT | NOT NULL]
[:=EXPRESSION];

Variable Name is the name of the valid variable type is the valid PL / SQL data types, and expression is the initial value of the variable.

If CONSTANT key word is present in the variable declaration, the variable must be initialized and its value can't be changed from this initial value. A constant variable is treated as Read only for the Remainder of the block.

If NOT NULL key word is present in the variable declaration then the variable must be initialized. If NOT NULL key word is not present in the variable declaration and value is not initialized, then it is assigned NULL by default.

Ex: BEGIN
INSERT INTO TEST VALUES(100,'RAJU',2000);
COMMIT;
END;

Ex: declare
    en number:=100;
    na char(10):='raju';
    sa number:=2000;
begin
    insert into test values(en,na,sa);
    commit;
end;
/

DBMS_OUTPUT

A built-in package that includes a number of procedures and functions that accumulates information in a buffer so that it can be retrieved later. These functions can also be used to display messages to the user.

PUT_LINE

It can also be used to display message to the user. PUT_LINE expects a single parameter of character data type. If used to display a message, it is the message string.

To display messages to the user, the SERVER OUTPUT should be set to on. It is a SQL * Plus environment parameter that displays the information passed as a parameter to the put_line function.

Ex: declare
    en number:=&eno;
    na char(10):=&name;
    sa number:=&salary;
begin
    dbms_output.put_line('employee no: '||en);
    dbms_output.put_line('employee name: '||name);
    dbms_output.put_line('employee salary: '||sa);
end;
/

SELECT

Type: PL/SQL

Syntax: SELECT COLUMN_LIST | * INTO VARIABLE_LIST | ROWTYPE FROM <TABLE> [WHERE <CONDITION>];

Select Column_list | * Into Variable_List | Rowtype From <Table> [where <Condition>];

This statement retrieves a row of data from particular table into PL / SQL variables.

COLUMN LIST

The database column name each one is separated by a comma. The * is short hand for the entire row (means all database column names).

VARIABLE LIST

Is the PL / SQL variable into which a select list item will go. Each PL / SQL variable should be compatible with its associated select list item (column list), and there should be the same number of select list items (column_list) and output variables (PL / SQL variables).

If the select list is '*', then this record could be defined as row type variable.

TABLE: Identifies the table from which to get the data.

WHERE CLAUSE: Is the criteria for the query.

NOTE: The form of the PL / SQL select statement described it should return no more than one row. The WHERE Clause will be compared against each row in the table. If it matches more than one row, PL / SQL will return an ERROR message.

Ex:

```
declare
        sa number;
begin
        select salary into sa from emp where eno=&eno;
        dbms_output.put_line('employee salary is '||sa);
end;
/
```

Dynamic Data Types (or) Composite Data Types

These data types hold a single value or more than one different value. PL / SQL introduce 2 types

1. Column Type **2. Row Type**

Column Type: VARIABLE TABLE.COLUMN %TYPE; → Variable table Column%.Type

Row Type: VARIABLE TABLE %ROWTYPE;

Variable Table%.Rowtype

Ex: EXAMPLE OF COLUMN TYPE

```
declare
        sa emp.sal%type;
begin
        select sal into sa from emp where eno=&eno;
        dbms_output.put_line(' employee salary is '||sa);
end;
/
```

Ex: EXAMPLE OF ROWTYPE

```
declare
        r emp%rowtype;
begin
        select * into r from emp where eno=&eno;
        dbms_output.put_line('employee name is '||r.name);
        dbms_output.put_line('employee job is '||r.job);
        dbms_output.put_line('employee salary is '||r.sal);
end;
/
```

PL / SQL CONTROL STRUCTURES

Like other 3rd generation languages PL / SQL has a variety of control structures that allow you to control the behavior of the block as it runs. These structures include conditional statements and loops.

The behaviour of IF statement is first it evaluates the condition, If it is true the next set of statements is executed. If the given condition is false the control will goes to the next elsif or else part.

1) SIMPLE IF

Syntax: IF CONDITION THEN

```
        <STATEMENT1>;
        <STATEMENT2>;
        ...
    END IF;
```

Ex: Accept any number to find out the absolute value without using Sql built-in function.

```
declare
        n number :=&no;
        k number:  -1;
begin
        if n<0 then
                n:= n*k;
        end if;
        dbms_output.put_line('absolute value of a given number..'||n);
end;
/
```

2) IF...ELSE

Syntax:

```
IF CONDITION THEN
        <SEQ_OF_STATEMENTS>;
ELSE
        <SEQ_OF_STATEMENTS>;
END IF;
```

Ex: WRITE A PL/SQL PROGRAM TO FIND OUT THE EMPLOYE BONUS BASED ON THE FOLLOWING CONDITIONS.

IF EMPLOYE JOB IS ='M' OR 'S'
BONUS IS 20% ON SALARY
OTHERWISE
BONUS IS 10% ON SALARY

Solution:

```
declare
        j emp.job%type;
        s emp.sal%type;
        b emp.bonus%type;
begin
        select job,sal into j,s from emp
        where eno=&eno;
        if upper(j)='m' or upper(j) ='s' then
                b:=s*20/100;
        else
                b:=s*10/100;
        end if;
        update emp set bonus b where eno &eno;
end;
```

update emp set bonus = b where eno = &eno;

3) LADDER IF

Syntax:
```
IF CONDITION THEN
    <SEQ_OF_STATEMENTS>;
ELSIF CONDITION THEN
    <SEQ_OF_STATEMENTS>;
[ELSIF CONDITION THEN
    <SEQ_OF_STATEMENTS>;]
END IF;
```

Or

```
IF CONDITION THEN
    <SEQ_OF_STATEMENTS>;
ELSE
    IF CONDITION THEN
        <SEQ_OF_STATEMENTS>;
    ELSE
        IF CONDITION THEN
            <SEQ_OF_STATEMENTS>;
        ...
        ELSE
            <SEQ_OF_STATEMENTS>;
        END IF;
    END IF;
END IF;
```

4) NESTED IF

```
IF <CONDITION> THEN
    IF <CONDITON>THEN
        SEQ_OF_STATEMETNS>;
    ELSIF <CONDITION> THEN
        SEQ_OF_STATEMENTS>;
    ...
    END IF;
ELSE
    SEQ_OF_STATEMENTS>;
END IF;
```

LOOP CONTROL STRUCTURES

PL / SQL provides a facility for executing statements repeatedly, via loops.
Loops are divided into 4 major Categories.

1. Simple Loop
2. Numeric For Loop
3. Cursor For Loop
4. WHILE Loop

Simple Loops

The most basic kind of loop is simple loop.

Syntax:

```
LOOP
    <SEQ_OF_STATEMETNS>
END LOOP;
```

In this loop the sequence of statements will be executed infinitely, since this loop has no stopping condition. However, we can add one with the exit statement.

EXIT

```
EXIT [WHEN <CONDITION>];
```

Ex:

```
declare
    i number:=1;
begin
    loop
        dbms_output.put_line('prs infosource');
        i:=i+1;
        exit when i>5;
    end loop;
end;
```

2) FOR (NUMERIC FOR)

Syntax:

```
FOR index in [REVERSE] <lower_limit>...<upper_limit>
    LOOP
        <SEQ_OF_STATEMENTS>;
    END LOOP;
```

INDEX: Is an implicitly declared integer whose value automatically increases or decreases by 1 on each iteration of the loop until the upper limit is reached.

REVERSE: Is a keyword, which causes the index to decrease by 1 for each iteration from the upper limit to the lower limit. Lower limit and upper limit are the loop range.

Ex:

```
declare
        r emp%rowtype;
begin
        for i in 1..5
        loop
                select * into r from emp where eno=i;
                dbms_output.put_line('employee salary :'||r.sal);
                dbms_output.put_line('employee job :'||r.job);
        end loop;
end;
```

2) WHILE LOOP

Syntax: WHILE <CONDITION>
LOOP
<SEQ_OF_STATEMENTS>;
END LOOP;

WHILE loop is used to execute set of statements repeatedly while the condition is TRUE.

Ex:

```
declare
        r emp%rowtype;
        i number;
begin
        while i<=5
                loop
                        select * into r from emp where eno=i;
                        dbms_output.put_line('employee salary :'||r.sal);
                        dbms_output.put_line('employee job :'||r.job);
                        i:=i+1;
                end loop;
end;
```

CURSORS

In order to process SQL statements, ORACLE will allocate an area of memory, known as the 'context area'. This area contains information necessary to complete the processing, including the Number of rows processed by the statement. A cursor is a handle, or pointer to the context area (SQL Private Area). Through the cursor a PL / SQL programme can control the context area and what happens to it as the statement is processed.

The ORACLE engine uses a work area for its internal processing in order to execute an SQL Statement. This work area is private to SQL's operations, and is called a cursor.

The data that is stored in the cursor is called the 'achieve data set'.

Types of Cursors:

Cursors are classified depending on the circumstances under which they are opened. If the ORACLE engine for its internal processing has opened a cursor they are known as implicit cursors.

A user can also open a cursor for processing data as required. Such user-defined cursors are known as explicit cursors.

Implicit Cursors

The ORACLE engine implicitly opens a cursor on the server to process each SQL statement. Since the implicit cursor is opened and managed by the ORACLE engine internally, THE FUNCTION OF RESERVING AN AREA IN MEMORY, POPULATING THIS AREA WITH APPROPRIATE DATA, PROCESSING THE DATA IN THE MEMORY AREA, RELEASING THE MEMORY AREA WHEN THE PROCESSING IS COMPLETE is taken care of by the ORACLE engine.

GENERAL CURSOR ATTRIBUTES

When the ORACLE engine creates implicit or explicit cursors, cursor control variables are also created to control the execution of the cursor. Whenever any cursor is opened and used, the ORACLE engine creates a set of 4system variables, which keeps the track of the current status of a cursor. The cursor variables can be accessed and used in a PL / SQL code block. Both implicit and explicit cursors have 4 attributes.

IMPLICIT CURSOR ATTRIBUTES

%FOUND

Syntax: SQL%FOUND

It is a Boolean attribute it returns either true or false. This attributes returns true, if an insert, update or delete affected one or more rows, or a single row select returned one or more rows. Otherwise it evaluates to false.

% NOTFOUND

Syntax: SQL%NOTFOUND

It is a Boolean operator, it returns true or false. It is the logical opposite of %FOUND.

% ROWCOUNT

Syntax: SQL%ROWCOUNT

This attribute returns the number of rows effected by an insert, update, or delete and select into statement.

% ISOPEN

Syntax: SQL%ISOPEN

The ORACLE engine automatically opens and closes the SQL cursor after executing its associated insert, delete, update or select statements has been processed incase of implicit cursors.

NOTE: SQL%ISOPEN always evaluates to false.

Ex: SQL%FOUND

```
begin
        delete emp where eno=&eno;
        if sql%found then
                dbms_output.put_line('records are deleted...');
        else
                dbms_output.put_line('records are not deleted...');
        end if;
end;
/
```

PROCESSING EXPLICIT CURSORS

When individual records in a table have to be processed inside a PL / SQL code block a cursor is used. This cursor will be declared and mapped to an SQL query in the declare section of the PL / SQL block.

A cursor is created and used is known as explicit cursor. The cursor declaration is the only step that goes in the declarative section of the PL / SQL block. The other 3 steps (open, fetch, close) are found in the executable or exception sections.

EXPLICIT CURSOR MANAGEMENT

The following steps involved in using an explicit cursor processing.

1. Declare a cursor mapped to a SQL select statement.
2. Open the cursor for a query.
3. Fetch the results or data from the cursor one row at a time into PL / SQL variables.
4. Close the cursor.

CURSOR DECLARATION

A cursor is defined in the declarative part of a PL / SQL block. The name of the cursor and associates it with a select statement.

Syntax: CURSOR <CURSOR_NAME> IS QUERY(SELECT STATEMENT);

Where cursor_name is the name of the valid cursor and query is a valid SQL select statement.

OPEN STATEMENT

OPEN <CURSOR_NAME>;

This statement is used to open the previously declared cursor.

FETCH STATEMENT

Syntax: FETCH <CURSOR_NAME>INTO PL / SQL LIST_OF_VARIABLES | ROWTYPE;

This statement is used to retrieve data from the currently opened cursor into list of PL / SQL variables.

CLOSE STATEMENT

Syntax: CLOSE <CURSOR_NAME>;

This statement is used to close the currently opened cursor.

EXPLICIT CURSOR ATTRIBUTES

1) %FOUND 2) %NOTFOUND 3) %ROWCOUNT 4) % ISOPEN

Ex: WRITE A PL/SQL PROGRAM TO FIND OUT ALL THE STUDENTS TOTAL MARKS, RESULTS, AND RESULTS.

Solution:

```
declare
        cursor st is select * from student;
        r st%rowtype;
begin
        open st;
        loop
                fetch st into r;
                exit when st%notfound;
                r.tm:=r.mvb+r.moracle;
                r.avm:=r.tm/2;
                if r.mvb>=35 and r.moracle>=35 then
                        if r.avm>=60 then
                                r.result:='first';
                        elsif r.avm>=50 then
                                r.result := 'second';
                        elsif r.avm>=35 then
                                r.result:='third';
                        end if;
                else
                        r.result:='fail';
                end if;
                update student set tm=r.tm,avm=r.avm,result=r.result
                where rollno=r.rollno;
        end loop;
        close st;
end;
/
```

CURSOR FOR

Syntax: FOR ROW TYPE – VARIABLE IN CURSOR_NAME
LOOP
STATEMENT;
END LOOP;

A cursor for loop automatically does the following tasks.

1. Implicitly declares a row type variable.
2. Opens a cursor.
3. Fetches a record from the currently opened cursor.
4. Close the cursor when all rows have been processed.

Ex:

```
declare
        cursor st is select * from student;
begin
        for r in st
        loop
                r.tm:=t.mvb+r.moracle;
                r.avm:=r.tm/2;
                if r.mvb>=35 and r.moracle>=35 then
                        if r.avm>=60 then
                                r.result:='first';
                        elsif r.avm>=50 then
                                r.result := 'second';
                        elsif r.avm>=35 then
                                r.result:='third';
                        end if;
                else
                        r.result:='fail';
                end if;
                update student set tm=r.tm,avm=r.avm,result=r.result
                where rollno=r.rollno;
        end loop;
end;
/
```

DYNAMIC CURSORS

Another name of dynamic cursor is 'parameterized cursor'.

Till now all the cursors that have been declared and used fetch a predetermined set of records. In other words, the criteria on which the active data set is determined is hard coded and never changes.

Syntax: CURSOR <CURSOR_NAME> (PARAMETER DATATYPE, - - -) IS QUERY;

ORACLE recognizes this and permits the creation of a parameterized cursor. Hence the contents of the opened cursor will constantly change depending upon a value passed.

Since the cursor accepts values, it is called parameterized cursor. The parameters can be either a constant or a variable.

OPEN STATEMENT

Syntax: OPEN <CURSOR_NAME> (PARAMETER |VARIABLE | VALUE, - - -);

This statement is used to open a cursor to the given variable values.

Ex:

```
declare
        cursor a(en number) is select * from cust where eno=en;
        r cust%rowtype;
        t number:=0;
begin
        open a(100);
        loop
                fetch a into r;
                exit when a%notfound;
                t:=t+r.amount;
        end loop;
        dbms_output.put_line('bill amount '||t);
end;
/
```

EXCEPTIONS

Any well program must have the ability to handle the errors. PL / SQL implements error handling with exceptions and exception handlers. Exceptions can be associated with ORACLE errors or user defined errors.

PL / SQL programs able to deal with both expected and unexpected errors during execution of a PL / SQL block.

ERROR TYPES

1. COMPILE TIME ERRORS
2. RUNTIME ERRORS

COMPILE TIME ERRORS

These error reports and you have to correct them.

RUNTIME ERRORS

These errors are raised and caught by exception handlers.

EXCEPTION TYPES

ORACLE supports two types of exceptions.

1. Predefined Exceptions

ORACLE has predefined exceptions that corresponding to the most common errors. These errors are defined in the package standard, because of this they are already available to the program, but it is not necessary to declare in the declarative section like a variables or user defined exceptions.

Exception	Description
NO_DATA_FOUND	A PL / SQL SELECT RETURNS NO DATA.
TOO_MANY_ROWS	A PL / SQL SELECT STATEMENT RETRIEVES MORE THAN ONE RECORD.
DO_VAL_ON_INDEX	UNIQUE CONSTRAINT VIOLETED
VALUE_ERROR	TRUNCATION, ARITHMATIC ERROR.
ZERO_DIVIDE	VALUE WAS DIVIDED BY ZERO
INVALID_NUMBER	CONVERSION TO A NUMBER FAILED
INVALID_CURSOR	ILLEGAL CURSOR OPERATION
CURSOR_ALREADY_OPEN	ATTEMPT TO OPEN A CURSOR i.e. ALREADY OPENED

1. Syntax:

```
EXCEPTION
WHEN <EXCEPTION> THEN
SEQ_OF_STATEMENTS;
[WHEN <EXCEPTION_NAME> THEN
SEQ_OF_STATEMENTS;]
[WHEN OTHERS THEN
SEQ_OF_STATEMENTS;]
```

2. Syntax:

```
EXCEPTION
WHEN <EXCEPTION_NAME> Or <EXCEPTION_NAME>
THEN
SEQ_OF_STATEMENTS;
[WHEN OTHERS THEN
    SEQ_OF_STATEMENTS;]
```

Then exception section consists of handlers for all the exceptions. An exception handler contains the code i.e. executed when the associated error with the exception occurs and the particular Exception is raised. Each exception handler consists of when clause and statements to execute when the exception is raised.

OTHERS

Exception handler will execute for all raised exceptions. It should always be the last handler in the PL / SQL block.

Ex:

```
declare
        n number :=&eno;
        r emp%rowtype;
begin
        select * into r from emp where eno=n;
        dbms_output.put_line('employee job is '||r.job);
exception
        when no_data_found then
        dbms_output.put_line('employee no not found sor y');
end;
/
```

SQLCODE AND SQLERRM

PL / SQL provides two built-in functions SQL CODE AND SQL ERRM. SQLCODE returns the current error code, and SQLERRM returns the current SQL error message. The maximum length of error message is 512 characters.

Note: The values of SQLCODE and SQLERRM are assigned to local variables. First, then these variables are used in SQL statements.

USER DEFINED EXCEPTIONS

A user-defined exception is an error that is defined by the programmers. The error that signifies is not necessarily an ORACLE error, it could be an error with the data.

User defined exceptions are declared in the declarative section of a PL / SQL block just like variables. Exceptions have a type "EXCEPTION".

RAISE

User defined exceptions are raised explicitly via the RAISE statements, while predefined exceptions are raised implicitly when their associated ORACLE error occurs.

Ex:

```
declare
        my exception;
        n number;
        ina varchar2(10):=&iname
begin
        select stock into n from item_stock where iname=ina;
        if n>=100 then
                update item_stock set stock=stock-20 where iname=ina;
        else
                raise my;
        end if;
        commit;
        select  stock  into n from item_stock where iname=ina;
        dbms_output.put_line(' current stock is '||n);
        exception  when my then
        dbms_output.put_line(' insufficient stock....please check it..');
end;
/
```

DATABASE TRIGGERS

Triggers are similar to procedures or functions. Like procedures and functions triggers must be stored in the database and cannot be local to a block. A trigger is executed implicitly whenever the triggering event happens and a trigger does not accept arguments. The act of executing the trigger is known as firing the trigger. The triggering event is a DML operation on a database table (insert, update, delete).

Triggers can be used for many things.

1. Maintaining complex integrity constraints not possible through declarative constants enabled at table creation.
2. Auditing information in a table.
3. Automatically signaling other programs that action needs to take place when changes are made to a table.

Trigger Event or Statement

It is a SQL statement that causes a trigger to be fired. It can be insert, delete, and update statement for a specific table.

Trigger Restriction

A trigger restriction specifies a Boolean expression that must be true for the trigger to fire. It is an option available for triggers that are fired for each row. A trigger restriction is specified by using a when clause.

TRIGGER ACTION

A trigger action is the pl/sql code to be executed when a triggering statement is encountered and any triggering restriction evaluates to true.

Syntax: CREATE OR REPLACE TRIGGER <TRIGGER_NAME>
[BEFORE|AFTER] TRIGGERING EVENT ON
<TABLE_NAME>FOREACH ROW [WHEN <COND>]
TRIGGER_BODY;

This command is used to create a new database trigger, where TRIGGER_NAME is the name of the trigger TRIGGERING EVENT specifies any *DMLC statement (insert/delete/update). TABLE NAME is the name of the table and the TRIGGER BODY is the valid pl/sql code for this trigger. The triggering condition in the WHEN clause, if present, is evaluated first. The body of the trigger is executed only when this condition evaluates to true.

MAIN COMPONENTS OF A TRIGGER

The required components of a trigger are:

1. Trigger_Name
2. Triggering_Event
3. Trigger_body

NOTE: OLD AND NEW is called as pseudo records or bind variables.

RAISE APPLICATION ERROR:

This function is used to create your own error messages.

Syntax: Raise_Application_Error (error no, error message);

Where error number is a parameter that should be between 20,000 to 20,999 where error message is the text that associated with this error.

Ex: NOT NULL (Trigger for not null)

```
cl scr
create or replace trigger no_con before insert on emp_test for each row
begin
        if :new.empno is null; then
        raise_applicatin_error (-20,000, 'null vaues are not allowed');
end if;
end;
/
```

RESTRICTIONS ON TRIGGERS

1. A trigger may not issue any TCLC (commit, rollback) statements.
2. The trigger body can't declare any long or long raw variables.

DROPING AND DISABLING TRIGGERS

DROP TRIGGER

Syntax: DROP TRIGGER <TRIGGER_NAME>;

This command is used to remove the database trigger from the database.

ALTER TRIGGERS

Syntax: ALTER TRIGGER <TRIGGER_NAME> DISABLE | ENABLE;

This command is used to disabling or enabling the existed database trigger.

DISABLE

This clause is used to deactivate the existed database triggers.

ENABLE

This clause is used to activate the disabled trigger.

DISABLE ALL TRIGGERS

This clause is used deactivate all the database triggers.

ENABLE ALL TRILGGERS

This clause is used to activate the all disabled triggers.

Ex:

```
cl scr
        create or replace trigger b_udi before insert or delete or update on
        emp_res for each row
declare
        code char;
begin
        if inserting then
                code := 'i';
        elsif updating then
                code :='u';
        elsif deleting then
                code:='d';
        end if;

        if to_char (sysdate, 'hh') >=6 and code='i' then
                raise_application_error (-20000,'invalid time to insert data...
                sorry');
        elsif to_char (sysdate, 'hh')>=6 and code='u' then
                raise_application_error (-20000,'invalid time to update data...
                sorry');
        elsif to_char (sysdate, 'hh') and code='d' then
                raise_application_error (-20890,'invalid time to delete data...
                sorry');
        end if;
end;
/
```

INSTEAD OF TRIGGERS

Pl / Sql 8.0 provides one additional kind of trigger is instead of triggers. these triggers can be defined on views.

USING TRIGGERS (INSTEAD) DELETING RECORDS FROM VIEW.

Ex:

```
cl scr
        create or replace trigger b_ins_del instead of delete on each row
begin
        delete from sales_info where slno =:old.slno;
end;
```

INSTEAD OF UPDATE

Ex:

```
sql> create or replace trigger b_ins_del instead of update on sal_repo on
     each row
begin
     update sales_info set samt = :new.samt where slno = :old.slno;
end;
/
```

TO DISPLAY ALL THE TRIGGERS FROM CURRENT DATA DICTIONARY
Syntax: SELECT TRIGGER_NAME FROM USER_TRIGGERS;

TO DISPLAY THE ALL TRIGGERS ON THE SPECIFIED DATABASE TABLE
Syntax: SELECT TRIGGER_NAME, TRIGGERING_EVENT FROM USER_TRIGGERS WHERE TABLE_NAME='<TABLE_NAME>';

FUNCTIONS

FUNCTION is a subprogram, which returns one and only one value. Function has a return clause associated.

Or

A function is logically grouped set of SQL and PL / SQL statements that performs a specific task. A stored function is a named PL / SQL code block that have been compiled and stored in ORACLE engines system table.

The function must set the value of the return parameter and be of the same data type specified in the return clause of the function definition. Multiple return statements are also allowed, but only one will be executed by any one call.

A function is made up of a declarative part, executable part and exception handling part. The functions code can be stored in the database.

However a procedure call is a PL / SQL statement by itself, while a function call is called as part of an expression.

HOW THE ORACLE ENGINE CREATES A FUNCTION

When a function is created, the ORACLE engine automatically performs the following tasks.

1. Compiles the function code.
2. Stores the function code in the database.

The ORACLE engine compiles the PL / SQL code block. If an error occurs the ORACLE engine displays a message after creation that the function was created with compilation errors.

TO SEE THE CURRENT FUNCTION ERRORS INFORMATION

```
show error
        or,
select * from user_errors;
```

ADVANTAGES OF FUNCTIONS

1. SECURITY
2. PERFORMANCE
3. MEMORY ALOOCATION
4. PRODUCTIVITY
5. INTEGRITY

1. SECURITY

Stored functions can help enforce data security.

2. PERFORMANCE

It improves data base performance in the following ways.

a) Amount of information sent over a network is less.

b) Once the function is present in the SGA (System Global Area) retrieval from disk is not required every time different users call the same function.

Memory Allocation: The amount of memory used reduces as stored functions have shared memory capabilities. Only one copy of the function needs to be loaded for execution by multiple users.

Productivity

By writing functions redundant coding can be avoided, and increasing productivity.

Integrity

A function needs to be tested only once to guarantee that it returns accurate result.

Syntax: CREATE [OR REPALCE] FUNCTION FUNCTION_NAME> (PARAMETER [IN] TYPE [DEFAULT VALUE | EXPRESSION] . . .) RETURN TYPE IS | AS FUNCTION_BODY;

Where function_name is the name of the function, parameter and type are name of the variable and data type. Return type is return data type is the type of the value that the function returns and function body is a PL / SQL block containing the code for the function.

RETURN STATEMENT

Syntax: RETURN VALUE | EXP;

Inside the body of the function statement is used to control the calling environment with a value.

Ex: CALLED FUNCTION TO FIND BIG

```
create or replace function big(a in number,b in number)
return number is
begin
        if a>b then
                return a;
        else
                return b;
        end if;
end;
/
```

Ex: CALLING FUNCTION

```
declare
        fno number :=&fno;
        sno number:=&sno;
        b number;
begin
        b:=big(fno,sno);
        dbms_output.put_line('biggest digit is ' b);
end;
```

PROCEDURES

PL / SQL procedures behave very much like procedures in other 3rd generation languages. They share many of the same properties.

Procedure is a subprogram it accepts one or more arguments and it returns one or more values.

POINTS TO REMEMBER

1. When a procedure is created, first it is compiled then stored in the database in compiled form. This compiled code can be run later, from another PL /SQL block or execute command.

2. When the procedure is called, parameters can be passed.

3. Procedure is to call as part of an expression

4. A procedure is a PL / SQL block with a declarative, executable and exception handling sections.

Syntax:
```
CREATE [OR REPALCE] PROCEDURE <PROG_NAME>
[(ARGUMENT [IN|OUT] TYPE, . . .)]
{IS | AS}
PROCEDURE_BODY;
```

Where procedure name is a name of the procedure to be created, argument is the name of a procedure parameter or variable, type is the data type of the associated parameter (data type) and procedure body is a PL / SQL block that makes up the code of the procedure.

IN

Specifies that a value for the argument must be specified when calling the procedure. By default if takes IN.

OUT

Specifies that the procedure pass a value for this argument back to its calling environment after execution.

Ex: TO FIND OUT THE BIGGEST NUMBER (CALLED PROCEDURE)

```
create or replace procedure big (a in number, b in number, b_num out number)
is
begin
        if (a>b) then
                b_num:=a;
        else
                b_num: =b;
        end if;
end;
```

Ex: CALLING PROCEDURE

```
declare
        a number :=&fno;
        b number :=&sno;
        big number;
begin
        big (a,b,big);
        dbms_output.put_line ('biggest '||big);
end;
```

PACKAGES

Packages are the 3rd type of named PL / SQL blocks after procedures and functions. They are very useful feature for PL / SQL.

A package is a PL / SQL construct that allows related objects to be stored together. A package has two separate parts.

1. Package Specification or Package Header 2. Package Body

Package Specification: The package specification is also known as the package header, it contains the information about the contents of the package. However it does not contain the code for any procedures or functions.

Syntax:

```
CREATE [OR REPLACE] PACKAGE <PACKAGE_NAME>{IS
|AS}
PROCEDURE . . . | FUNCTION | . . . EXCEPTIONS, . . .
END;
```

Package Body: The package body contains the code of the functions and procedures of the package.

Syntax: **CREATE [OR REPLACE] PACKAGE <PACKAGE_NAME>**
body
{IS | AS}
PROCEDURE code. . . | FUNCTION code| . . .
END;

Ex: PACKAGE HEADER

```
create or replace package pfp is
        procedure big_num(a in number,b in number,res out number);
        function big(a in number,b in number) return number is;
end;
```

Ex: PACKAGE BODY

```
create or replace package body pfp is
        procedure big_num(a in number,b in number,res out number)
        is
        begin
                if a>b then
                        res:=a;
                else
                        res:=b;
                end if;
        end;

        function big(a in number,b in number) return number is
        s number;
        begin
                if a>b then
                        s:=a;
                else
                        s:=b;
                end if;
                return s;
        end;
end;
```

ORACLE 8 LOB's

LOB DATA TYPES

The lob types are used to store large objects either a binary or character value up to 4 gigabytes in size. Large objects can contain unstructured data, which is accessed more efficiently than long or long raw data types, with fewer restrictions. Lob types are manipulated using the dbms_lob package.

Or

Oracle8 has to support LOB's a useful feature of any relational database. to store large amount of data, either text or binary in nature.

A lob is simply a database field that holds a large amount of data, such as a graphic file or long text document. In oracle 7.x, a varchar2 field can hold up to 2000 bytes.

A long or long raw column can hold up to 2 gigabytes, but there are many restrictions on long data type, such as the fact that there can be only one long column per a table etc... The interface for manipulating long or long raw data piecewise is the oci(oracle call interface).

Oracle8 provides a solution. LONG and LONG RAW columns are still available with the same restrictions, but there is new datatype family available is LOB family. There are four different kinds of LOB's, which are designed for different kinds of data

1) clob 2) blob 3) nclob 4) bfile

clob, blob, nclob's are collectively known as internal lob's.
Bfile is known as external lobs.

Characteristics of a lob

The maximum size of a LOB is 4gb, rather than the oracle7 limit of 2gb for long and long raw.

LOB's can be manipulated using either the oracle8 call interface (OCI) or through pl/sql with dbms_lob package. Both of these interfaces provide random access to a lob type for both reading and writing except for BFILE.

The restrictions on long or long raw data do not apply to LOB's.
lob's can be used as bind variables.

Lobs are manipulated using either sql dmlc statements or pl/sql dbms_lob package.

Data type	description
1) clob	This data type is similar to the oracle 7.x long type, a clob can hold 4 gigabytes.
2) blob	This data type is similar to the oracle7.x long raw type, a blob can hold 4 GB.
3) nclob	An nclob stores multibyte national character set data.
4) bfile	BFILE allow read-only access to large binary files stored outside the oracle database.

lob storage

Unlike long or long raw data, lob data is not stored inline in a database table. Rather it is stored in a separate location, and a lob locator is stored in the original table.

Ex:

```
create table test_lob
(
        slno number(3),
        story_clob clob
);
```

```
insert into test_lob values(1,'lfdsldjfdl l');
insert into test_lob values(2,'fdsdfdfd');
```

test_lob

slno	story_clob
1	
2	

Initializing a lob column

A lob column can be set to NULL

Ex: insert into test_lob values(100,null,null);

in this case, a null is inserted in the column, rather than a lob locator. There is no actual storage allocated for the lob data, since there is no locator to point to it. Because of this, you cannot use dbms_lob on a NULL value. The row must be updated to a valid locator first.

One way of doing this is with the empty_clob( ) and empty_blob( ) functions.

dbms_lob package

Using sql you can manipulate the entire lob, you cannot manipulate pieces of the lob data. The dbms_lob package rectifies this by allowing piecewise manipulation (read and write for internal lob's).

Dbms_lob routines

The following routines are used to read lob values or return information about a lob value.

compare
getlength
read
instr
substr

The following routines are used to write to lob values.

append
copy
erase
trim
write

READ

Syntax: dbms_lob.read(lob variable, no_char_read, start_point_read, output_var);

To use the read procedure, you need to know the locator value of the lob you want to read. The locator value must be selected from the table that contains the lob. Since the locator value must be supplied as input to the read procedure, you should use pl/sql variables to hold the locator value. The read procedure, in turn, will place its output in a pl/sql variable. You can use the dbms_output package to display the output value.

```
create table test_clob
(
        slno number(4),
        story clob
);
```

```
insert into test_clob values(10,'oracle was developed by larry elison. the first name
                                of oracle corporation is RSI');

insert into test_clob values(11,'sql is a non-procedural language...');
```

Ex:

```
cl scr
set serveroutput on
set veri off
declare
        a clob;
        nc number:=&nc;
        output varchar2(35);
        scp number:=&scp;
begin
        select story into a from test_clob where SLNO=&SLNO;
        dbms_lob.read(a,nc,scp,output);
        dbms_output.put_line('story '||output);
end;
```

SUBSTR

This routine within the dbms_lob package performs the sql substr function on a lob value. This routine has three input parameters, which must be specified in this order

Syntax: output_var dbms_lob.substr(lob_var,no_chars,start_point_char);

Where lob_var lob locator
Where no_chars the number of characters
Where start_point_char starting point of the read

Ex:

```
cl scr
set serveroutput on
set veri off
declare
        a clob;
        nc number:=&nc;
        output varchar2(35);
        scp number:=&scp;
 begin
        select story into a from TEST_CLOB where SLNO=&SLNO;
        output:=dbms_lob.substr(a,nc,scp);
        dbms_output.put_line('story '||output);
 end;
 /
```

instr

This routine within the dbms_lob package performs the sql instr function on a lob value.

This routine has four input parameters, which must be specified in this order.

Syntax: output_var=dbms_lob(lob_var,pattern_var,start_point_read, occur_var);

Where lob_var is lob locator.
Where patter_var the pattern to be tested for clob's.
Where start_point_read is the starting point of the read of a lob value
Where occur_var the occurence of the pattern within the lob value.

Ex:

```
cl scr
set serveroutput on
set veri off
declare
        a clob;
        pchar varchar2(10):=&check_char;
        pos_char number;
```

```
        scp number:=&scp;
        occu_pos integer:=&ochar;
    begin
        select story into a from TEST_CLOB where SLNO=&SLNO;
        pos_char:=dbms_lob.instr(a,pchar,scp,occu_pos);
        dbms_output.put_line('story '||pos_char);
    end;
```

getlength

This function returns the current length of the specified lob. This function is semilar in function to the sql length function, but it is used only for lob values.

Note: you cannot use the sql length function on lob values.

Syntax: out_var=dbms_lob.getlength(lob_var);

Ex:

```
set serveroutput on
declare
        a clob;
        noc number;
begin
        select story into a from TEST_CLOB where SLNO=&SLNO;
        noc:=dbms_lob.getlength(a);
        dbms_output.put_line('lengh of a lob value..'||noc);
end;
/
```

compare

This routine of dbms_lob package compares two lob values. If the two lob values are the same it returns zero. Otherwise it returns a non-zero. This function has five input parameters.

Syntax: out_var=dbms_lob.compare(first_lob,second_lob,no_char,
first_start_point,second_start_point);

Where first_lob is the lob locator
Where second_lob lob locator
Where no_char is the number of characters to compare
Where first_start_point is the starting point of the first lob value
Where second_start_point is the starting point of the second lob value

Ex:

```
set veri off
set serveroutput on
cl scr
declare
        flob clob;
        slob clob;
        nc number:=&nc;
        fscp number:=&fscp;
        sscp number:=&sscp;
        out_char number;
begin
        select story into flob from modem where eno=&eno;
        select story into slob from modem where eno=&eno;
        out_char:=dbms_lob.compare(flob,slob,nc,fscp,sscp);
        dbms_output.put_line(out_char);
        if out_char=0 then
                dbms_output.put_line('two lob values are equal');
        else
                dbms_output.put_line('two lob values are not equal');
        end if;
end;
```

write

dbms_lob package allows you to write data in specific locations of the lob. You can write character data into clob fields using write procedure. This procedure has four input parameters, which must be specified in this order.

1) Lob locator for the lob.
2) Number of characters to write
3) Starting point of the write
4) The input text is assigned to a clob column text

Syntax: dbms_lob.write(lob_locator,no_char,start_char_point,upd_text);

Ex:

```
cl scr
set serveroutput on
declare
        a clob;
        scp number:=&scp;
        nc number;
```

```
        in_text varchar2(100);
    begin
        in_text:='this is new text';
        select story into a from TEST_CLOB where SLNO=&SLNO;
        nc:=length(in_text);
        dbms_lob.write( a,nc,scp,in_text);
    end;
    /
```

Ex2:

```
    cl scr
    set serveroutput on
    declare
        a clob;
        scp number;
        nc number;
        n number;
        in_text varchar2(100);
    begin
        in_text:='this is new text';
        select story into a from modem where eno=&eno;
        n:=dbms_lob.getlength(a);
        nc:=length(in_text);
        dbms_lob.write( a,nc,n,in_text);
    end;
```

append

Syntax: **dbms_lob.append(dest_lob_locator,sour_lob_locator);**

This routine is used to appends data from one lob to a second lob.

Ex:

```
    cl scr
    set serverouput on
    declare
        sa clob;
        db clob;
    begin
        select story into sa from modem where eno=&eno for update;
        select story into db from modem where eno=&eno for update;
        dbms_lob.append(db,sa);
    end;
```

ERASE

Syntax: dbms_lob.erase(lob_locator,no_char,start_char_pos);

This procedure is used to erase the contents of the lob.

Parameters are: -

1) Lob locator for the lob
2) No. of characters to erase
3) Starting point

Ex:

```
cl scr
set serveroutput on
set veri off
declare
        a clob;
        nc number:=&noc;
        spos number := &scp;
begin
        select story into a from test_clob where slno=&slno;
        dbms_lob.erase(a,nc,spos);
end;
```

copy

Syntax: dbms_lob.copy(dest_lob,sour_lob,no_char,dest_scp,sour_scp);

This procedure is used to copy from one lob into another lob.

Parameters are: -

1) Destination lob locator
2) Source lob locator
3) Number of characters to copy
4) To start writing to within the destination lob value
5) To start reading from within the destination

Copy is a combination of the read and write capabilities.

Note: Before that the specified record must be locked. pl/sql block must com
commit command.

Ex:

```
cl scr
set serveroutput on
declare
        dlob clob;
        slob clob;
        nc number :=&noc;
        dscp number :=&dscp;
        sscp number :=&sscp;
begin
        select story into dlob from test_clob where slno=&slno for update;
        select story into slob from test_clob where slno=&slno for update;
        dbms_lob.copy(dlob,slob,nc,dscp,sscp);
        commit;
end;
```

TRIM

This procedure is used to reduce the size of a lob's value by trimming characters from the end of the value (like sql rtrim function). When you specify the locator value for the lob and lob's new length.

Ex:

```
set serveroutput on
set veri off
cl scr
declare
        a clob;
        nlen integer;
begin
        nlen:=&noc;
        select story into a from test_clob where slno=&slno for update;
        dbms_lob.trim(a,nlen);
        commit;
end;
```