STRUCTURE QUERY LANGUAGE

Dual:

Dual is a dummy table and it is a data dictionary table.

Example 1
SELECT 1+2 FROM DUAL;
Output
1+2
3

Example 2
SELECT 'Hello',1+NULL,3 FROM DUAL;
Output
‘HELLO’ 1+2 3
Hello 3 3

Example 3
SELECT * FROM DUAL;
Output
DUMMY
X

Null:

Null is a unassigned value or unknown value or undefined value.
Null is not equal to zero.
Null is not equal to space.
Null is not equal to underscore.
Null is not equal to Null.
Null is not equal to ‘’
0 + NULL Null
0 - NULL Null
0 * NULL Null
0 / NULL Null
Null is always greater than any number because in ascending order Null comes last and in descending order it comes first or it is maximum in ascending order & minimum in descending order.

Example

To insert null values into table

```
INSERT ALL
INTO T99 VALUES('')
INTO T99 VALUES(NULL)
SELECT * FROM DUAL;
```

Data Types

Data type	Min size	Max size	Default size/Default format
Number(p.s)	Precision is 1 Scalar is -84	Precision is 38 Scalar is 127	
char	1 byte	2000 byte	1
nchar	1 byte	2000 byte	1
varchar2	1 byte	4000 byte	
nvarchar2	1 byte	4000 byte	
date	01-January-4712 BC	31-December-9999	dd-mom-yy
long	2 gigabytes		
blob	4 gigabytes		
clob	8 gigabytes		
timestamp			
bfile			

Long

1) A table can contain only one LONG column.
2) LONG column cannot appear in where clause, group by clause, order by clause and distinct.
3) A stored function cannot return a LONG value.
4) LONG column cannot be indexed.
5) If a table has LONG and LOB columns, then you cannot bind more than 4000 byte of data for both LONG and LOB. If a table has either LONG or LOB then we can bind more than 4000 byte of data.
6) We cannot set UNIQUE and PRIMARY KEY for LONG column.

Blob

The abbreviation is binary large object. It is used to store images (i.e. .gif, .jpeg, … etc.)

Clob

The abbreviation is character large object. It is used to store text files (i.e. .xls, .xml, .doc, … etc.¬)

Data Dictionary Tables

1) DICTIONARY – It is used to find list of data dictionary tables available in oracle.
2) USER_OBJECTS - It contains list of objects present in the database (i.e., table, index, view, function, type, trigger, package, package body, lob, procedure, function, materialized view, sequence…etc.)
3) USER_TABLES – It contains list of tables and global temporary tables present in the database.
4) USER_VIEWS - It contains list of views present in the database.
5) USER_ERRORS – It contains the error occurred during creating object like procedure, package body, function... etc.
6) USER_TAB_COLUMNS – It contains list of columns present in the entire table in the database.
7) USER_CONSTRAINTS – It contains the list of constraints which was created by all users in the database.
8) USER_INDEXES – It contains the list of index created by all users in the database.
9) USER_TRIGGERS - It contains the list of triggers present in the database.
10) USER_SOURCE - It contains the entire script on object like procedure, package, package body, function, trigger and type.
11) USER_MVIEWS – It contains the list of materialized view in the database.
12) USER_SEQUENCES - It contains the list of sequence present in the database.
13) USER_CONS_COLUMNS
14) ALL_USERS – It contains list of users who can login into database.

Table Partitioning

Table partition is a method of splitting an object (table or index) into separate partitions based on some criteria. The criteria can be date, number or character.

Example

Partitioned on character

```
CREATE TABLE SALES_LIST(
SALESMAN_ID NUMBER(5),
SALESMAN_NAME VARCHAR2(30),
SALES_STATE VARCHAR2(20),
SALES_DATE DATE)
PARTITION BY LIST(SALES_STATE)(
PARTITION SALES_WEST VALUES('MUMBAI','PUNE','GOA','NAGPUR','SURAT'),
PARTITION SALES_EAST
VALUES('KOLKATA','PATNA','RANCHI','JORHAT','AGARTALA'),
PARTITION SALES_NORTH
VALUES('DELHI','KANPUR','LUCKNOW','AGRA','MERUT'),
PARTITION SALES_SOUTH
VALUES('CHENNAI','HYDREBAD','COCHIN','BANGALORE','NELLORE')
);
```

Partitioned on date

```
CREATE TABLE SALES_RANGE(
SALESMAN_ID NUMBER(5),
SALESMAN_NAME VARCHAR(30),
SALES_AMOUNT NUMBER(5),
SALES_DATE DATE)
PARTITION BY RANGE(SALES_DATE)(
PARTITION SALES_JAN2000 VALUES LESS THAN
(TO_DATE('02/01/2000','MM/DD/YYYY')),
PARTITION SALES_FEV2000 VALUES LESS THAN
(TO_DATE('03/01/2000','MM/DD/YYYY'))
);
```

If table contains more number of rows then queries takes more time to execute a query and table becomes harder to manager. So in that case we use table partitioning to overcome those issues. We can partition Table, Index & Materialized View.

Global Temporary Tables

1) Data in temporary table is stored in temp segments in the temp table space.
2) Data in temporary tables is automatically deleted at the end of the database session, even if it ends abnormally.
3) Views can be created against temporary tables and combinations of temporary and permanent tables.
4) Temporary tables can have triggers associated with them.
5) Indexes can be created on temporary tables. The content of the index and the scope of the index is the same as the database session.

Syntax

To create a global temporary table

```
CREATE GLOBAL TEMPORARY TABLE table name (
  Column1  data type,
  Column2 data type
);
```

To drop a global temporary table.

```
DROP TABLE GLB_TEM;
```

ON COMMIT PRESERVE ROWS

This indicates that data should be present till the session ends

Example

```
CREATE GLOBAL TEMPORARY TABLE GLB_TEM (
  ENO  NUMBER,
  GENDER CHAR(2)
)ON COMMIT PRESERVE ROWS;
```

ON COMMIT DELETE ROWS

This indicates that data should be deleted at the end of this transaction. 'ON COMMIT DELETE ROWS' is the default one.

Example

```
CREATE GLOBAL TEMPORARY TABLE GLB_TEM (
  ENO  NUMBER,
  GENDER CHAR(2)
)ON COMMIT DELETE ROWS;
```

Real Time Example

In real time global temporary table is used to store a dataset that is used in most of the queries.

Materialized Views

Materialized view is a database objects that store the result of a query. It is a schema object. It can be called as snapshots. Materialized view occupies space in database.

Example

To create a materialized view

```
CREATE MATERIALIZED VIEW M1 AS
SELECT EMPLOYEE_ID, FIRST_NAME,SALARY FROM EMPL WHERE
EMPLOYEE_ID BETWEEN 100 AND 150;
```

To drop a materialized view

```
DROP MATERIALIZED VIEW M5;
```

To refresh a particular materialized view

```
EXECUTE DBMS_MVIEW.REFRESH('M1');
```

(or)

```
BEGIN
DBMS_MVIEW.REFRESH(M1,'C');
END;
/
```

To refresh entire materialized view

Read only materialized view

User cannot perform DML operations in read only materialized view.

Example

```
CREATE MATERIALIZED VIEW M30 AS
SELECT A,B FROM T1;
```

Updatable materialized view

User can perform DML (Insert, Update, and Delete) in updatable materialized view. The changes made using DML operation in materialized view will not be changed in base table.

Example

CREATE MATERIALIZED VIEW M30 FOR UPDATE AS
SELECT A,B FROM T1;
Writable materialized view
User can perform DML (Insert, Update and Delete) in writable materialized view. The changes made using DML operation in materialized view will not be changed in base table. If you refresh a materialized view then the changes made will not be updated in base table but it gets lost in materialized view itself.

Types of materialized view

1) Primary Key materialized view

This is default materialized view. When you create this type of materialized view then the base table must have primary key. Sub query can also be used in creating materialized view and it is called as sub-query materialized view.

2) Object materialized view

This is based on object table and created by using 'OF TYPE' clause.

3) ROWID materialized view

This is created based on rowed of the rows in a base table.

Example

CREATE MATERIALIZED VIEW M30 REFRESH WITH ROWID AS
SELECT * FROM T5;

4) Complex materialized view

Materialized view is consider as complex when you do restrictions by using DISTINCT, CONNECT BY, UNIQUE, INTERSECT, MINUS, UNION ALL, joins, aggregate functions… etc.

Materialized view keywords

1) ON COMMIT

Whenever you save pending changes then you materialized view will be refreshed automatically for this we use 'ON COMMIT'.

Example

CREATE MATERIALIZED VIEW LOG ON COURSE;
CREATE MATERIALIZED VIEW M5 REFRESH FAST ON COMMIT AS
SELECT * FROM COURSE;

2) ON DEMAND

When you use ‘ON DEMAND’ then materialized view must be refreshed every time by user. This is default one.

Example

```
CREATE MATERIALIZED VIEW M7 REFRESH ON DEMAND AS
SELECT * FROM COURSE;
execute DBMS_MVIEW.REFRESH('M7');
```

3) REFRESH FAST

This check for any changed made in log if yes then it refreshes a materialized view or else it doesn’t refresh

Example

```
CREATE MATERIALIZED VIEW LOG ON S1;
CREATE MATERIALIZED VIEW M22 REFRESH FAST AS
SELECT * FROM S1;
```

4) REFRESH COMPLETE

In this when you update materialized view after updating base table; the entire data in materialized view are deleted and then insert operation starts in materialized view.

Example

```
CREATE MATERIALIZED VIEW M22 REFRESH COMPLETE AS
SELECT ROWID AS R,CNAME FROM S1;
```

5) REFRESH FORCE

This method performs fast refresh if possible otherwise it performs complete refresh.

Example

```
CREATE MATERIALIZED VIEW LOG ON S1;
CREATE MATERIALIZED VIEW M22 REFRESH FORCE AS
SELECT ROWID AS R,CNAME FROM S1;
```

6) NEVER REFRESH

If you don’t want to refresh materialized view then use ‘NEVER REFRESH’

Example

```
CREATE MATERIALIZED VIEW M8 NEVER REFRESH AS
SELECT * FROM COURSE;
```

Materialized View LOG

Materialized view logs are used to track changes (insert, update and delete) to a table.

Purging materialized view logs

It show no of changes made to a table. If you refresh materialized view then this gets to zero.

SELECT COUNT(*) FROM MLOG$_S1 ;

Advantages

1) You can create materialized view on another materialized view. So that it reduces more network traffic.
2) It can be either read only, writable or updatable.
3) You can create materialized view for join queries and quires that contain aggregate functions.
4) It reduces network traffic.

Real Time Example

This is used when more number of user visit particular table repeatedly which will make network traffic. In this scenario materialized view will be created from where user can get data.

Index

It is used to improve the performance of query retrieval.

Index can be either Unique or Non-Unique index

Unique Index

It is created automatically when a user defines a primary/unique constraint for a column.

Non-unique Index

Created by the user to speed up the retrieval of rows. Maintained by oracle server

When to create index

1. When a table is large.

2. When most of the queries are expected to retrieve less than 4% of rows from the table.
3. When a column contains a wide range of values.
4. When column is most often used as a condition in a query.
5. When a column contains large number of null values.

When not to create index

1. When a table is small.
2. When most of the queries are excepted retrieve more than 4% of rows from the table.
3. When column contains a large number of duplicates.
4. When column is not most often used as condition in a query.
5. When a table is updated frequently.

Bitmap Index

Created for low cardinality column that contains less than 100 unique records. Create it when you have lot of duplicates.

Example

To create a bitmap index

```
CREATE BITMAP INDEX IX1 ON STUDENT(CID);
```

To drop a bitmap index

```
DROP INDEX IX1;
```

B-tree Index

Created for high cardinality column that contains more than 100 unique records.

Example

To create a index

```
CREATE INDEX IX1 ON STUDENT(CID);
```

To drop a index

```
DROP INDEX IX1;
```

Function based Index

Example

To create function based index

CREATE INDEX IX1 ON STUDENT(LOWER(SNAME));
CREATE BITMAP INDEX IX1 ON STUDENT(LOWER(SNAME));
To drop function based index
DROP INDEX IX1;

Composite Index

Example
To create composite index
CREATE INDEX IX1 ON STUDENT(CID,SNAME);
CREATE BITMAP INDEX IX1 ON STUDENT(CID,SNAME);
To drop composite index
DROP INDEX IX1;

Hints (/*+ */)

This is used to force the query to run particular index or full table scan.
Example
To run full table scan
SELECT /*+ FULL(STUDENT) */ SNAME FROM STUDENT WHERE SNAME='BRUCE';

Parsing

Oracle needs to execute some task before executing SQL query (which is received from user). This process is called
Parsing.

1) Syntactic check

Oracle parses the syntax to check for misspelled SQL keywords.

2) Semantic check

Oracle verifies all table names & column name from the data dictionary table and check to see if user is authorized to view data.

3) Optimization

Oracle creates an execution plan on schema statistics.
If the entire above task performs then it is called hard parsing. If only syntactic check & semantic check perform then it is called soft parsing. Soft parsing occurs when you use ‘cursors’.

Synonyms

A synonym is an alternative name for object such as tables, view, sequences, stored procedures and other database objects. You can create synonym for synonym. It is object of a database.

To view list of synonyms and which object does it affect

SELECT * FROM USER_SYNONYMS;

There are two types of synonyms. They are

i) Private Synonym

This synonym is accessed within the schema. The default one is private.

Example

To create private synonym

CREATE SYNONYM STUDENT1 FOR STUDENT;

To drop private synonym

DROP SYNONYM STUDENT1;

ii) Public Synonym

Example

To create public synonym

CREATE PUBLIC SYNONYM STUDENT1 FOR STUDENT;

To drop public synonym

DROP PUBLIC SYNONYM STUDENT1;

Sequence

It automatically generates unique number. It is a schema object. It replaces the application code. It is

Mainly used for generating primary key column values.

Example

To create sequence

Create SEQUENCE seq

START with 10

INCREMENT by 2

MAXVALUE 22;

To alter sequence

Alter SEQUENCE seq

INCREMENT by 1

MAXVALUE 30;
To drop sequence
Drop SEQUENCE seq;

List of clauses for sequence

1) INCREMENT BY – It specifies the interval between sequence numbers. The value can be either positive number or negative number but not zero. If the value is positive the sequence ascends and if the value is negative than sequence descends. If you omit this clause then the default increment value will be 1.

2) MAXVALUE – It is used to specify the maximum value that a sequence can generate. This MAXVALUE must be greater than or equal to START WITH and must be greater than MINVALUE.

3) START WITH – It specifies first sequence number to be generated. It can be ascending or descending.

4) MINVALUE – It specifies the minimum value of the sequence. This MINVALUE must be lesser than or equal to START WITH and must be lesser than MINVALUE.

5) NOMAXVALUE – It specifies a maximum value of 10 powers 27 for ascending order or -1 for descending order.

6) NOMINVALUE – It specifies a minimum value of 1 for ascending order and -10 powers 26 for descending order.

7) CYCLE – It specifies that when the sequence continues to generate when it is reached maximum or minimum. Once if the sequences value reaches maximum then it generates the minimum value and if the sequence value reaches minimum then it generates the maximum value.

8) NOCYCLE – It specifies that when the sequence cannot generate more valued if it is reached maximum or minimum value. This is the default one.

9) CACHE – It indicates that the value of the sequence is pre-allocated and keeps in memory for faster access. The integer value can have 28 or fewer digit. The minimum value of the parameter is 2.

10) NOCACHE – It indicates that value of sequence is not pre-allocated. If you leave CACHE and NOCACHE then the database allocated for 20 sequence numbers by default.

11) ORDER – To sequence number are generated in the order of request. This clause is very useful when you are using sequence numbers as timestamps.
12) NOORDER – To specify sequence number are not generated in order of request. This is default.

Example 1:-

```
Create SEQUENCE seq
NOMINVALUE
INCREMENT by 2
NOMAXVALUE;
```

Example 2:-

```
Create SEQUENCE seq
NOMINVALUE
MAXVALUE 22;
```

Example 3:-

```
Create SEQUENCE seq
START with 1
INCREMENT by 2
NOMAXVALUE;
```

Example 4:-

```
Create SEQUENCE seq
MINVALUE 2
INCREMENT by 2
NOMAXVALUE;
```

Example 5:-

```
Create SEQUENCE seq
START WITH 2
INCREMENT BY 1
MAXVALUE 10
NOCACHE
CYCLE;
```

Example 6:-

```
Create SEQUENCE seq
START WITH 1
INCREMENT BY 1
```

MAXVALUE 10
CACHE 10;
Example 7:-
CREATE SEQUENCE seq
START WITH 1
INCREMENT BY -1
MAXVALUE 10;

Views

View is virtual table. It logically represents a subset of data from one or more table.
We can create a view from another view

Advantages

1) To make more complex query look simple.
2) To restrict data access.

Example

To create a view

```
CREATE VIEW V1 AS
SELECT S.SNAME, C.CNAME, F.FNAME FROM  COURSE C, STUDENT S, FACULTY F
WHERE S.CID=C.CID AND C.CID=F.CID;
```

To alter a view

```
CREATE OR REPLACE VIEW V1 AS
SELECT * FROM T1;;
```

To drop a view

```
DROP VIEW V1;
```

To view query of the view

```
SELECT * FROM USER_VIEWS;
```

To view errors of a view

```
SELECT * FROM USER_ERRORS;
```

Read Only

This can be used when you don't want to manipulate data in view.
Example

```
CREATE OR REPLACE VIEW V1 AS
SELECT A FROM T1 WHERE A=1 WITH READ ONLY;
```

Updatable View

Here you can insert, update & delete records but this action only be performed on base table or a table from which the view is created.

Example

```
CREATE OR REPLACE VIEW V1 AS
SELECT SNAME FROM STUDENT WHERE CID = 10;
```

Check Option

This is used when you want to insert data in views by checking condition from where clause. If it true then insert operation takes place.

Example

```
CREATE OR REPLACE VIEW V1 AS
SELECT A FROM T1 WHERE A IN (1,3) WITH CHECK OPTION;
```

Force View

We can create a view with a dummy base table which does not exist in a database. This view will be having errors

Example

```
CREATE OR REPLACE FORCE VIEW V1 AS
SELECT * FROM GREENS;
```

Restrictions

- If you are using pseudo column like ROWNUM, ROWID and LEVEL then you must use alias.
- If you are using group functions then you must use alias.

For updatable view

- You can't use ORDER BY and GROUP BY.
- You should not use analytical and aggregate function.
- Sub query in SELECT list

Constraints

Constraints are of two types. They are

i) Table Level Constraints – The constraints can be specified after all the columns are defined. This is called table-level definition.

ii) Column Level Constraints - The constraints can be specified immediately after the column definition. This is called column-level definition.

1) Primary key

It is a unique identifier.
Ignores null values,
It is entity integrity.
It will not accept duplicate values.
Only one primary key is allowed in a table.
It automatically generates unique index.

Example

To add a primary key using 'create' command

```
CREATE TABLE WORKERS(
WMNO VARCHAR2(6),
WMNAME VARCHAR(20),
WMDES VARCHAR(20),
CONSTRAINT X1 PRIMARY KEY(WMNO)
);
```

To add a primary key using 'create' command without using constraint name

```
CREATE TABLE WORKERS(
WMNO VARCHAR2(6) PRIMARY KEY,
WMNAME VARCHAR(20),
WMDES VARCHAR(20)
);
```

To add a primary key using 'alter' command

```
ALTER TABLE WORKERS ADD CONSTRAINT X1 PRIMARY KEY(WMNO);
```

To drop a primary key which has constraint name

```
ALTER TABLE WORKERS DROP CONSTRAINT X1;
```

To enable a primary key

```
ALTER TABLE WORKERS ENABLE CONSTRAINT X1;
```

To disable a primary key

```
ALTER TABLE WORKERS DISABLE CONSTRAINT X1;
```

2) Unique key

It is unique identifier.

It accepts null values and can accept more than 1 null value.
It is entity integrity.
It automatically generates unique index.
Example
To add unique key using ‘create’ command with constraint name

```
CREATE TABLE WORKERS(
WMNO VARCHAR2(6),
WMNAME VARCHAR(20),
WMDES VARCHAR(20),
CONSTRAINT X1 UNIQUE(WMNO)
);
```

To add unique key using ‘create’ command without constraint name

```
CREATE TABLE WORKERS(
WMNO VARCHAR2(6) UNIQUE,
WMNAME VARCHAR(20),
WMDES VARCHAR(20)
);
```

To drop unique key using ‘alter’ command

```
ALTER TABLE WORKERS ADD CONSTRAINT X1 UNIQUE(WMNO);
```

To drop unique key

```
ALTER TABLE WORKERS DROP CONSTRAINT X1;
```

3) Foreign key

It is referential integrity.
It refers a primary (or) unique constraint of another table.
It accepts null values.
It accepts duplicate values.
Example
To create foreign key using ‘create’ command from primary key

```
CREATE TABLE WORKERS(
WMNO VARCHAR2(6),
WMNAME VARCHAR(20),
WMDES VARCHAR(20),
CONSTRAINT X1 PRIMARY KEY(WMNO)
);
CREATE TABLE LOC(
```

WMNO VARCHAR2(6),
WLN NUMBER(20),
CONSTRAINT X2 FOREIGN KEY(WMNO) REFERENCES WORKERS(WMNO)
);
To create foreign key using ‘create’ command from unique key
CREATE TABLE WORKERS(
WMNO VARCHAR2(6),
WMNAME VARCHAR(20),
WMDES VARCHAR(20),
CONSTRAINT X1 UNIQUE(WMNO)
);
CREATE TABLE LOC(
WMNO VARCHAR2(6),
WLN NUMBER(20),
CONSTRAINT X2 FOREIGN KEY(WMNO) REFERENCES WORKERS(WMNO)
);
To create foreign key using ‘alter’ command
ALTER TABLE LOC ADD CONSTRAINT X2 FOREIGN KEY(WMNO) REFERENCES WORKERS(WMNO)
To drop foreign key
ALTER TABLE LOC DROP CONSTRAINT X2;

4) Not null
It is domain integrity.
Ignores null values.
Example
To add not null using ‘create’ command
CREATE TABLE WORKERS(
WMNO VARCHAR2(6) NOT NULL,
WMNAME VARCHAR(20),
WMDES VARCHAR(20)
);
To add not null using ‘alter’ command
ALTER TABLE WORKERS MODIFY WMNO VARCHAR2(6) NOT NULL;
To drop not null
ALTER TABLE WORKERS MODIFY WMNO VARCHAR2(6) NULL;

5) Check

It is domain integrity.
It should satisfy a condition.
Example
To add check constraint using ‘create’ command with constraint name

```
CREATE TABLE WORKERS(
WMNO VARCHAR2(6),
WMNAME VARCHAR(20),
WMDES VARCHAR(20),
WSAL NUMBER(8),
CONSTRAINT X1 CHECK(WSAL BETWEEN 10000 AND 20000)
);
```

To add check constraint using ‘create’ command without constraint name

```
CREATE TABLE WORKERS(
WMNO VARCHAR2(6),
WMNAME VARCHAR(20),
WMDES VARCHAR(20),
WSAL NUMBER(8) CHECK(WSAL BETWEEN 10000 AND 20000)
);
```

To add check constraint using ‘alter’ command

```
ALTER TABLE WORKERS ADD CONSTRAINT X1 CHECK(WSAL BETWEEN 10000
AND 20000);
```

To drop check constraint.

```
ALTER TABLE WORKERS DROP CONSTRAINT X1;
```

When you use number

```
CHECK(WSAL BETWEEN 10000 AND 20000)
```

When you use character

```
CHECK(WLOC='CHENNAI');
X1 CHECK(WMDES IN ('CHENNAI','HYDREBAD'))
```

When you use date

Example
To see list of columns names, data type, table name…etc in entire database

```
SELECT * FROM USER_TAB_COLUMNS;
```

To see list of constraints names, constraint types, table name…etc in entire database
SELECT * FROM ALL_CONSTRAINTS;

Constraint types in ALL_CONSTRAINTS

Primary key	P
Foreign key	R
Unique	U
Not null	C
Check	C

Data Control Language

1) GRANT

This command is used to assign a privilege to the user. You can assign SELECT, INSERT, UPDATE, DELETE, REFERENCES, ALTER and INDEX privilege to user.
Syntax
GRANT PRIVILEGE ON OBJECT TO USER
Example
To grant single privilege
GRANT SELECT ON EMPLOYEES TO HR
To grant multiple privilege
GRANT SELECT, INSERT, DELETE ON EMPLOYEES TO HR
To grant all privilege
GRANT ALL ON EMPLOYEES TO HR
To grant all privilege to public (all users)
GRANT ALL ON EMPLOYEES TO PUBLIC
You can give privilege on functions & procedures
Syntax
To grant privilege on functions or procedures
GRANT EXECUTE ON OBJECT TO USER

2) REVOKE

This command is used to revoke the privileges assigned to the user.
Syntax
REVOKE PRIVILEGE ON OBJECT FROM USER

Example
To revoke single privilege
REVOKE SELECT ON T1 FROM HR
To revoke multiple privilege
REVOKE SELECT, INSERT, DELETE ON T1 FROM HR
To revoke all privilege
REVOKE ALL ON T1 FROM HR
To revoke all privilege to public (all users)
REVOKE ALL ON EMPLOYEES FROM PUBLIC

Syntax
To revoke privilege on functions or procedures
REVOKE EXECUTE ON OBJECT FROM USER

Transmission Control Language

1) COMMIT

Saves all pending changes permanent.
Example
COMMIT;

2) SAVEPOINT

It is a marker. It is used to identify a point to which you can roll back later. If two save point have same name then when you rollback will done until the last occurrence of the save point.
Example
SAVEPOINT A;

3) ROLLBACK

Discard all pending changes
Example
To discard all pending changes
ROLLBACK;
To discard to particular save point
ROLLBACK TO B;

4) SET TRANSACTION

It is used to set current transaction to read only and read write. This dosen't affects other transaction by other users. To end this transaction use 'COMMIT' or 'ROLLBACK'. Name is the transaction name given by user.

Example 1

```
SET TRANSACTION READ ONLY NAME 'TORENTO';
```

Example 2

```
SET TRANSACTION READ WRITE NAME 'TORENTO';
```

Data Manipulation Language

These command used to manage data within schema objects.

1) SELECT

This command is used to retrieve data from the database

Example

```
SELECT * FROM T1;
```

2) INSERT

This command is used to insert data into a table.

Example

To insert into table with specifying column name

```
INSERT INTO T1 (A) VALUES(6);
```

To insert into table without specifying column name

```
INSERT INTO T1 VALUES(5);
```

3) UPDATE

This command is used to update data in a table.

Example

To update entire row in a table

```
UPDATE T1 SET A=8;
```

To update particular row in a table

```
UPDATE T1 SET A=7 WHERE A=6;
```

4) DELETE

This command is used to remove one or more rows from a table. It can be rollback.
Example
To delete all rows from a table
DELETE FROM T1;
To delete one row from a table
DELETE FROM T1 WHERE A=4;

5) MERGE

Merge is an update plus insert. It will be updated when it is matched and will be inserted when it is unmatched.
Example

```
MERGE
INTO T2
USING T1
ON (T1.A = T2.A)
WHEN MATCHED
THEN UPDATE
SET T2.B = T1.B
WHEN NOT MATCHED
THEN
INSERT (T2.A,T2.B) VALUES(T1.A,T1.B);
```

Data Definition Language

These commands help you to manage the database structure. When you execute DDL commands then auto commit takes place at the end of transaction. The lists of DDL Commands are.

1) CREATE

This command is used to create database and database objects.
Example
To create database

```
CREATE DATABASE DB1;
```

To create objects in the database

```
CREATE TABLE L1(
A NUMBER(4)
```

);

2) DROP

This command is used to drop database and database objects.
Example
To drop database
DROP DATABASE DB1;
To drop objects in the database
DROP TABLE L1;

3) ALTER

This allows you to rename an existing table and It can also be used to add, modify, or drop a column from an existing table.
Example
To add a column to a table
ALTER TABLE L1 ADD B VARCHAR2(4) NOT NULL;
To drop a column from a table
ALTER TABLE L1 MODIFY B NUMBER(5);
To modify a column from a table
ALTER TABLE L1 DROP COLUMN B;
To rename a database object
ALTER TABLE L1 RENAME TO L2;

4) RENAME

This command is used to rename object in the database.
Example
RENAME L60 TO T1;

5) TRUNCATE

It deletes all data from the table. Actually truncate doesn't remove data but de-allocates whole Data pages and pointers to index.
Example
TRUNCATE TABLE L1;

- It is much faster.
- We can't use condition if a statement contains Truncate.
- A trigger doesn't fire on truncate.

- It cannot be rolled back.

6) COMMENTS

Example

To create comment on column

```
COMMENT ON COLUMN EMPL.FNAME IS 'MAIDON NAME';
```

To alter comment on column

```
COMMENT ON COLUMN EMPL.FNAME IS 'MAIDON NAME IS FATHER NAME';
```

To drop comment on column

```
COMMENT ON COLUMN EMPL.FNAME IS '';
```

Joins

Selecting data from two or more table is called joins. When we use alias it is much faster. Types of join are

1) Equi Join

To select matched rows from two or more table is called equi join. It can be called as inner join or natural join (natural join find common columns from both the table).

Example

Option 1

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S,
COURSE C
WHERE
S.CID = C.CID;
```

Option 2

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S INNER JOIN COURSE C
ON
S.CID = C.CID;
```

Option 3

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S JOIN COURSE C
ON
S.CID = C.CID;
```

Option 4

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S NATURAL JOIN COURSE C;
```

2) Outer Join

a) Left Outer Join

To get matched record from both the table and unmatched record from left table.

Example

Option1

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S,
COURSE C
WHERE
S.CID = C.CID(+);
```

Option2

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S LEFT OUTER JOIN COURSE C
```

```
ON
S.CID = C.CID;
```

b) Right Outer Join

To get matched record from both the table and unmatched record from right table.

Example

Option1

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S,
COURSE C
WHERE
S.CID(+) = C.CID;
```

Option2

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S RIGHT OUTER JOIN COURSE C
ON
S.CID = C.CID;
```

c) Full Outer Join

To get both matched and un-matched record from both the table.

Example

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S FULL OUTER JOIN COURSE C
ON
S.CID = C.CID;
```

3) Cross Join

It is a Cartesian product. Number of rows in first table is joined with number of rows in the second table. Cartesian product is formed when user ignores 'WHERE' clause. Cross join is used by developers to perform performance testing.

Example

```
SELECT
S.SNAME,
C.CNAME
FROM
STUDENT S,
COURSE C;
```

4) Self-Join

Joining a table with itself is called self-join.

Example

```
SELECT
E1.FIRST_NAME ENAME,
E2.FIRST_NAME MNAME
FROM
EMPLOYEES E1,
EMPLOYEES E2
WHERE
E1.MANAGER_ID = E2.EMPLOYEE_ID;
```

5) Semi Join

It returns row from first table when one or more row matched found in second table. Difference between semi-join and inner join is that row in first table will be returned only once even if the second table contains two matches from a row in the first table, only one copy of the row will be returned. Semi-joins are written by using 'IN' & 'EXIXTS' constructs.

Example

Option 1

```
SELECT C.CID, C.CNAME
FROM COURSE C
```

WHERE C.CID IN (SELECT DISTINCT(S.CID) FROM STUDENT S WHERE S.CID IS NOT NULL);
Option 2

6) Anti Join

It returns row from first table when no matches found in second table. Anti-joins are written by using 'NOT IN' & 'NOT EXISTS' constructs.
Example
Option 1

```
SELECT C.CID, C.CNAME
FROM COURSE C
WHERE C.CID NOT IN (SELECT DISTINCT(S.CID) FROM STUDENT S WHERE S.CID IS NOT NULL);
```

Option 2

Set Operators

There are four set operator in SQL. They are UNION, UNION ALL, INTERSECT and MINUS.
Let us take two tables T1 and T2

T1	T2
1	1
2	2
3	3
4	5

1) UNION

Displays data from both tables eliminating duplicate rows.
Example

```
SELECT A FROM T1
UNION
SELECT A FROM T2;
```

2) UNION ALL

Displays data from both tables without eliminating duplicate rows and it is faster.
Example

```
SELECT A FROM T1
```

UNION ALL
SELECT A FROM T2;

3) INTERSECT

Displays data which is commonly present both the table.
Example

```
SELECT A FROM T1
INTERSECT
SELECT A FROM T2;
```

4) MINUS

Displays data from Table 1 which is not present in Table 2.
Example

```
SELECT A FROM T1
MINUS
SELECT A FROM T2;
```

Rules

- It works from top to bottom.
- Both column data type should be same.
- Number of columns in both the query must be same.

Sub Queries

A query embedded within another query is called sub query. There are two types of sub query they are

a) Single Row Sub query

When a sub query returns one row then it is called single row sub query.
Example

```
SELECT FIRST_NAME, SALARY FROM EMPLOYEES WHERE SALARY > (SELECT
SALARY FROM
EMPLOYEES WHERE FIRST_NAME='Neena');
```

b) Multi Row Sub query

When a sub query returns more than one row is called multi row sub query.
Example

```
SELECT FIRST_NAME, SALARY FROM EMPLOYEES WHERE SALARY >ALL
```

(SELECT SALARY FROM EMPLOYEES WHERE FIRST_NAME='David');
Scalar Sub query
Whenever a ‘Select’ clause followed by sub query is called scalar sub query
Example
Whenever a SELECT 1+2+(SELECT 3+4 FROM DUAL) FROM DUAL;

Inline view

Whenever a ‘From’ clause followed by sub query is called Inline view or named sub query in a from clause is called Inline view.
Example
SELECT FIRST_NAME FROM (SELECT FIRST_NAME FROM EMPLOYEES
WHERE DEPARTMENT_ID=90);

Nested Sub query

Whenever a ‘Where’ clause followed by sub query is called nested sub query. Here first inner query is executed then outer query will be executed. It doesn’t have any reference between inner query and outer query. Inner query runs only once for outer query.
Example
SELECT MAX(SALARY) FROM EMPLOYEES
WHERE SALARY < (SELECT MAX(SALARY) FROM EMPLOYEES);

Co-related Sub query

When the inner query has reference to outer query is called correlated sub query. Here outer query runs first and then inner query is executed. It is user row by row processing. The inner query is executed once for every row of the outer query. It is faster.
Example
SELECT SALARY FROM EMPLOYEES E1 WHERE (3-1) =
(SELECT COUNT(DISTINCT(E2.SALARY)) FROM EMPLOYEES E2
WHERE E2.SALARY > E1.SALARY);

Rules for writing sub query

- Sub query must be enclose in brackets
- Sub query cannot be used after order by

- When we use multi-row sub query then we need to use relational operators like IN, ALL & ANY.

Pseudo Columns

1) SYSDATE

This command is used to display current system date in DD-MOM-YY

Example
SELECT SYSDATE FROM DUAL;

2) SYSTIMESTAMP

This command is used to display current system date & time.
Example
SELECT SYSTIMESTAMP FROM DUAL;

3) USER

This command shows you the current user you have logged in.
Example
SELECT USER FROM DUAL;

4) UID

This command shows you the current user id you have logged in.
Example
SELECT UID FROM DUAL;

5) ROWNUM

This command is used to generate row number starting from 1 and will not be stored in the database.
Example
SELECT ROWNUM FROM EMPLOYEES;

6) ROWID

ROWID is unique and generated when the row is inserted. It has 18 digit characters (contains mixed or character & numbers). This command is user to display row id.

Example
SELECT ROWID FROM EMPLOYEES;

7) NEXTVAL

This command is used to show next available value from sequence which can be used.
Example
SELECT SEQ10.NEXTVAL FROM DUAL;

8) CURRVAL

This command shows the current value from the sequence that has been already used.
Example
SELECT SEQ10.CURRVAL FROM DUAL;

9) LEVEL

Example
SELECT EMPLOYEE_ID, FIRST_NAME, MANAGER_ID, LEVEL AS STAGE FROM EMPLOYEES START WITH MANAGER_ID IS NULL CONNECT BY PRIOR EMPLOYEE_ID=MANAGER_ID ORDER BY STAGE ASC

SQL Functions

Conditional Functions or Control Statements

1) Case

Example
```
SELECT DEPARTMENT_ID,
CASE
WHEN DEPARTMENT_ID=10 THEN 'A'
WHEN DEPARTMENT_ID=20 THEN 'B'
ELSE 'X' END
FROM DEPARTMENTS;
```

2) Decode

This command is similar to if.. then.. else.. Statement. If the default statement is omitted then null will be used if no matches found.

Syntax

DECODE(COLUMN NAME, SERACH, RESULT, SEARCH, RESULT, DEFAULT)

Example

SELECT DEPARTMENT_ID, DECODE(DEPARTMENT_ID,10,'MANAGER',20,'ASST MANAGER',30,
'PROJECT MANAGER',40,'TEAM LEADER','OTHERS') FROM DEPARTMENTS;

Comparison Functions

1) Greatest

This command returns the greatest value in the arguments list. This command can contain one or more arguments.

Example

SELECT GREATEST(EMPLOYEE_ID, MANAGER_ID, DEPARTMENT_ID, SALARY, COMMISSION_PCT) FROM EMPLOYEES;

2) Least

This command returns the least value in the arguments list. This command can contain one or more arguments.

Example

SELECT LEAST(EMPLOYEE_ID, MANAGER_ID, DEPARTMENT_ID, SALARY, COMMISSION_PCT) FROM EMPLOYEES;

General Functions

1) NVL

It accepts exactly two arguments. If the first argument is null then display the second argument else display the first argument.

Example

SELECT NVL(COMMISSION_PCT,0)FROM EMPLOYEES;

2) NVL2

It accepts exactly three arguments. If the first argument is null then display the third argument else display the second argument.

Example

SELECT NVL2(COMMISSION_PCT,2,1)FROM EMPLOYEES;

3) NULLIF

It accepts exactly two arguments. Display null if both the arguments are same else display the first argument

Example

SELECT NULLIF(SALARY,24000) FROM EMPLOYEES;

4) COALESCE

It can contain two or more number of arguments and it returns the first not null from the arguments list.

Example

SELECT COALESCE(COMMISSION_PCT,SALARY) FROM EMPLOYEES;

Aggregate Functions or Group Functions

1) AVG

This command is used to retrieve average.

Example

SELECT AVG(SALARY) FROM EMPLOYEES;

2) COUNT

This command is used to display the count.

COUNT (COLUMN_NAME) – displays number of not null column in particular column.

COUNT (*) – displays number of row in the table without eliminating null columns or rows.

Example

SELECT COUNT(SALARY) FROM EMPLOYEES;

3) MAX

This command is used to retrieve maximum number or salary.

Example

SELECT MAX(SALARY) FROM EMPLOYEES;

4) MIN

This command is used to retrieve minimum number or salary.
Example
SELECT MIN(SALARY) FROM EMPLOYEES;

5) SUM

This command to retrieve by summing up all the rows in particular column. It works only if the column data type is number.
Example
SELECT SUM(SALARY) FROM EMPLOYEES;

There are certain rules for writing aggregate functions.

i) Group functions accept only one argument.
ii) Group functions ignore null values including COUNT(*)
iii) Whenever column is followed by group function or group function followed by column should have 'group by'
iv) These group functions SUM & AVG only work in number and not in date and character.
v) The '*' cannot be used except COUNT.
vi) When you are restricting with group functions use 'HAVING' clause.

Number Functions

1) ROUND

This command is used to round the decimal values. If the decimal value is greater than or equal to .5 then it is rounded to next value else rounded to current value.
Example
SELECT ROUND(157.7), ROUND(157.5), ROUND(157.3) FROM DUAL;
Output
ROUND(157.4) ROUND(157.5) ROUND(157.7)
157 158 158

2) TRUNC

This command is used to round the decimal point to current value.
Example
SELECT TRUNC(157.7), TRUNC(157.5), TRUNC(157.4) FROM DUAL;

Output
TRUNC(157.7) TRUNC(157.4) TRUNC(157.5)
157 157 157

3) MOD

This command is used to show the remainder.
Example
SELECT MOD(5,2),MOD(2,2) FROM DUAL;
Output
MOD(5,2) MOD(2,2)
1 0

Date Functions

1) MONTHS_BETWEEN

This command accepts exactly two arguments and it is used to show difference in months between two dates. If argument1 is greater than argument 2 then output will be in positive integer else output will be in negative integer.
Example
SELECT MONTHS_BETWEEN(SYSDATE,'22-AUG-13') FROM DUAL;

2) ADD_MONTHS

This command accepts exactly two arguments and it is used for adding months to current date.
Example
SELECT ADD_MONTHS(SYSDATE,12), ADD_MONTHS(SYSDATE,-12) FROM DUAL;

3) NEXT_DAY

This command accepts exactly two arguments and it is use to show on which date the next day comes.
Example
SELECT NEXT_DAY(SYSDATE,'FRIDAY') FROM DUAL;

4) LAST_DAY

This command accepts exactly two arguments and it is used to show last day of a month.
Example
SELECT LAST_DAY(SYSDATE) FROM DUAL;

Case Manipulation Functions

1) UPPER

This command can accept only one argument and displays in upper case format.
Example
SELECT UPPER(FIRST_NAME) FROM EMPLOYEES;

2) LOWER

This command can accept only one argument and displays in lower case format.
Example
SELECT LOWER(FIRST_NAME) FROM EMPLOYEES;

3) INITCAP

This command can accept one argument and display first character in capital.
Example
SELECT INITCAP(SNAME) FROM STUDENT;

Character Manipulation Function

1) CONCAT

This command accepts exactly two arguments and displays the result by concatenating the two arguments.
Example
SELECT CONCAT(FIRST_NAME,LAST_NAME) FROM EMPLOYEES;

2) LENGTH

This command can accept only one argument and displays the length of the string.
Example

SELECT LENGTH(FIRST_NAME) FROM EMPLOYEES;

3) REPLACE

This command can accept exactly two or three arguments and helps to replaces the character.
REPLACE(FIRST_NAME,'e','x') – Here ‘e’ is replaced with ‘x’.
REPLACE(FIRST_NAME,'e') – Here ‘e’ is removed.
Example
SELECT FIRST_NAME, REPLACE(FIRST_NAME,'e','x') FROM EMPLOYEES;
SELECT FIRST_NAME, REPLACE(FIRST_NAME,'e') FROM EMPLOYEES;

4) REVERSE

This command accepts exactly one argument and display the string in reverse order.
Example
SELECT REVERSE(SNAME) FROM STUDENT;

5) TRANSALATE

This command replaces sequence of string with another set of character. For example it replaces 1st character from string to replace with 1st character from replacement string then 2nd character from string to replace with 2nd character from replacement string so on.
Example
SELECT TRANSLATE('1ABC23','123','456') FROM DUAL;

6) LPAD

This command can accept exactly two or three arguments and
LPAD (SNAME,10,'@') – here if the string less than 10 characters then remaining character is filled by ‘@’ in left hand side and displays exact 10 characters.
LPAD(SNAME,5) – Here it display the string only up to 5 characters.
Example
SELECT LPAD(SNAME,10,'@') FROM STUDENT;
SELECT LPAD(SNAME,5) FROM STUDENT;

7) RPAD

This command can accept exactly two or three arguments and
RPAD (SNAME,10,'@') – here if the string less than 10 characters then remaining character is filled by '@' in right hand side and displays exact 10 characters.
RPAD(SNAME,5) – Here it display the string only up to 5 characters.
Example
SELECT RPAD(SNAME,10,'@') FROM STUDENT;
SELECT RPAD(SNAME,5) FROM STUDENT;

8) LTRIM
This command is used to remove special character or character on left hand side.
Example
SELECT LTRIM('****HELLO**WORLD****','*') FROM DUAL;

9) RTRIM
This command is used to remove special character or character on right hand side.
Example
SELECT RTRIM('****HELLO**WORLD****','*') FROM DUAL;

10) TRIM
This command is used to remove special character or character on both left & right hand side.
Example
SELECT TRIM('*' FROM '****HELLO**WORLD****') FROM DUAL;

11) SUBSTR
This command is used to get the string of particular length from particular starting position.
SUBSTR(SNAME,1,3) – It is used to print 3 character starting from position 1.
SUBSTR(SNAME,-3) – it is used to print 3 character starting from last position.
Example
SELECT SUBSTR(SNAME,1,3) FROM STUDENT;
SELECT SUBSTR(SNAME,-3) FROM STUDENT;

12) INSTR

This command can accept two to four arguments. It is used find the position of occurrence of a specified character.

Syntax

INSTR(STRING1, STRING 2, STARTING POSITION, Nth APPERANCE)

Example

```
SELECT INSTR('TECH ON THE NET','E') FROM DUAL;
SELECT INSTR('TECH ON THE NET','E',1,2) FROM DUAL;
```

13) ASCII

This command returns ASCII values for specified character. If you declare a word then it returns the ASCII values for 1st character.

Example

```
SELECT ASCII('C') FROM DUAL;
```

Analytical Functions

1) RANK

It provides rank to record based on some column value. In case if a tie of 2 record occurs at position N then the two record positions will be N and gives N+2 to the next record.

Example

```
SELECT EMPLOYEE_ID, SALARY, RANK() OVER(ORDER BY SALARY DESC) FROM
EMPLOYEES;
```

2) DENSE_RANK

It provides rank to record based on some column value. In case if a tie of 2 record occurs at position N then the two record positions will be N and gives N+1 to the next record

Example

```
SELECT EMPLOYEE_ID, SALARY, DENSE_RANK() OVER(ORDER BY SALARY DESC)
FROM
EMPLOYEES;
```

3) ROW_NUMBER

It gives a running serial number to a record.

Example
SELECT EMPLOYEE_ID, SALARY, ROW_NUMBER() OVER(ORDER BY SALARY DESC) FROM
EMPLOYEES;

4) LEAD

This command computes an expression on next row is return the value to the current row.

Example
SELECT FIRST_NAME,SALARY, LEAD(SALARY,1,0) OVER(ORDER BY SALARY DESC) FROM
EMPLOYEES;

5) LAG

This command computes an expression on previous row is return the value to the current row.

Example
SELECT FIRST_NAME,SALARY, LAG(SALARY,1,0) OVER(ORDER BY SALARY DESC) FROM
EMPLOYEES;

6) FIRST_VALUE

This command picks the first record from the partition after doing order by and first record are returned.

Example
SELECT EMPLOYEE_ID, HIRE_DATE, DEPARTMENT_ID,
FIRST_VALUE(HIRE_DATE)
OVER(PARTITION BY DEPARTMENT_ID ORDER BY HIRE_DATE) FROM
EMPLOYEES WHERE
EPARTMENT_ID IN(10,20,30,40,50);

7) LAST_VALUE

This command picks the last record from the partition after doing the order by and last record are returned.

Example

```
SELECT EMPLOYEE_ID, HIRE_DATE, LAST_VALUE(HIRE_DATE) OVER() FROM
EMPLOYEES
WHERE DEPARTMENT_ID IN(10,20,30,40,50) ORDER BY HIRE_DATE;
```

8) CUBE & ROLLUP

Both these command is used in 'group by' clause.

Example

```
SELECT DEPARTMENT_ID, SUM(SALARY) AS TOTAL FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (10,20,30,40) GROUP BY CUBE(DEPARTMENT_ID);
```

Output:-

-	58400
10	4400
20	19000
30	24900
40	6500

Example

```
SELECT DEPARTMENT_ID, SUM(SALARY) AS TOTAL FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (10,20,30,40) GROUP BY
ROLLUP(DEPARTMENT_ID);
```

Output:-

10	4400
20	19000
30	24900
40	6500
-	58400

Operators

a) Condition symbol (or) Relational operator (or) Comparison operator

=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Lesser than or equal to

<>(or)!= Not equal to
These operators can hold only single values after this conditional symbol.

Example 1
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY = 24000;
Example 2
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY > 17000;
Example 3
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY < 24000;
Example 4
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY >= 17000;
Example 5
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY <= 17000;
Example 6
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY <> 17000;
Example 7
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY != 17000;

b) Logical operator

i) AND
This command indicates that both the condition must satisfy.
Example
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE SALARY > 14000
AND SALARY < 24000;

ii) OR
This command indicates that either one of the condition must satisfy.

Example
SELECT EMPLOYEE_ID, FIRST_NAME FROM EMPLOYEES WHERE LAST_NAME = 'King'
OR LAST_NAME = 'Chen';

c) Relational operator
IN NOT IN
LIKE NOT LIKE
BETWEEN NOT BETWEEN
IS NULL IS NOT NULL

ANY
ALL

i) IN & NOT IN
This command can hold one or more values after 'IN' operator.
Example 1
SELECT FIRST_NAME FROM EMPLOYEES WHERE SALARY IN (17000, 24000);
Example 2
SELECT FIRST_NAME FROM EMPLOYEES WHERE SALARY NOT IN (17000, 24000);

ii) LIKE & NOT LIKE
'S%' – It will return a string starting with capital S.
'%S' – It will return a string ending with capital S.
'%S%' – it will return a string that is containing capital S(it can be at starting, middle or ending).
'500_%' – It will return a string that is started by '500'.
'%_08' – It will return a string that is ended by '08'.
Example 1
SELECT FIRST_NAME FROM EMPLOYEES WHERE FIRST_NAME LIKE 'S%';
Example 2
SELECT FIRST_NAME FROM EMPLOYEES WHERE FIRST_NAME NOT LIKE 'S%';

iii) BETWEEN & NOT BETWEEN

‘BETWEEN’ the minimum value or date must be first and then then the max value or date should follow.

Example 1

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY BETWEEN 2400 AND 5000;
```

Example 2

```
SELECT HIRE_DATE FROM EMPLOYEES WHERE HIRE_DATE BETWEEN '21-SEP-89' AND '21-MAY-91';
```

Example 3

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY NOT BETWEEN 2400 AND 5000;
```

Example 4

```
SELECT HIRE_DATE FROM EMPLOYEES WHERE HIRE_DATE
NOT BETWEEN '21-SEP-89' AND '21-MAY-91';
```

iv) IS NULL & IS NOT NULL

‘IS NULL’ command is used to get null value columns and ‘IS NOT NULL’ command is used to get not null value columns.

Example 1

```
SELECT COMMISSION_PCT FROM EMPLOYEES WHERE COMMISSION_PCT IS NULL;
```

Example 2

```
SELECT COMMISSION_PCT FROM EMPLOYEES WHERE COMMISSION_PCT IS NOT NULL;
```

v) ANY

>ANY (17000, 14000, 13500) – It retrieves salary greater than least number present in parenthesis.

<ANY (17000, 14000, 13500) – It retrieves salary lesser then greatest number present in parenthesis.

<>ANY (17000, 14000, 13500) (or) !=ANY (17000, 14000, 13500) – It retrieves salary lesser then , greater than and equal to value present in the parenthesis.

>=ANY (17000, 14000, 13500) – It retrieves the salary greater than or equal to the least value present in the parenthesis.

<=ANY (17000, 14000, 13500) – It retrieves the salary lesser than or equal to the greatest value present in the parenthesis.

=ANY (17000, 14000, 13500) – It retrieves the salary equal to the value present in the parenthesis.

Example 1

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY >ANY
(17000,14000,13500);
```

Example 2

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY <ANY
(17000,14000,13500);
```

Example 3

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY <>ANY
(17000,14000,13500);
```

Example 4

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY !=ANY
(17000,14000,13500);
```

Example 5

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY >=ANY
(17000,14000,13500) ;
```

Example 6

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY <=ANY
(17000,14000,13500);
```

Example 7

```
SELECT SALARY FROM EMPLOYEES WHERE SALARY =ANY
(17000,14000,13500);
```

vi) ALL

>ALL (17000, 14000, 13500) – It retrieves salary greater then greatest number present in parenthesis.

<ALL (17000, 14000, 13500) – It retrieves salary lesser then least number preset in parenthesis.

<>ALL (17000, 14000, 13500) (or) !=ALL (17000, 14000, 13500) – It retrieves salary lesser then least number in parenthesis and salary greater then greatest number present in parenthesis.

>=ALL (17000, 14000, 13500) – It retrieves salary greater than or equal to greatest number present in parenthesis.

<=ALL (17000, 14000, 13500) – It retrieves salary lesser than or equal to least number present in parenthesis.

Example 1
SELECT SALARY FROM EMPLOYEES WHERE SALARY >ALL (17000,14000,13500);
Example 2
SELECT SALARY FROM EMPLOYEES WHERE SALARY <ALL (17000,14000,13500);
Example 3
SELECT SALARY FROM EMPLOYEES WHERE SALARY <>ALL
(17000,14000,13500);
Example 4
SELECT SALARY FROM EMPLOYEES WHERE SALARY != ALL
(17000,14000,13500);
Example 5
SELECT SALARY FROM EMPLOYEES WHERE SALARY >=ALL
(17000,14000,13500);
Example 6
SELECT SALARY FROM EMPLOYEES WHERE SALARY <=ALL
(17000,14000,13500);

d) Arithmetic operator

+	Addition
-	Subtraction
*	Multiplication
/	Division

Order of precedence
Parenthesis
Multiplication
Division
Addition
Subtraction

CONVERSIONS

CLAUSES
These are the type of clauses are.
WHERE

This clause is used to restrict number of rows to return by a condition, for joining two or more tables, restricts views and materialized views.

HAVING

GROUP BY

This clause is used to group the results by one or more columns used in group by. It is most often used when there is aggregate or group functions are used in query. The column you are specifying in group by clause must be in select list same thing applies to sub query where column name should be present in the current scope of select list and not in outer query. You can use analytical function 'ROLLUP' and 'CUBE' in group by clause.

Example

```
SELECT FIRST_NAME, COUNT(*) AS NUM FROM EMPLOYEES GROUP BY
FIRST_NAME;
SELECT DEPARTMENT_ID, SUM(SALARY) AS TOTAL FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (10,20,30,40) GROUP BY CUBE(DEPARTMENT_ID);
SELECT DEPARTMENT_ID, SUM(SALARY) AS TOTAL FROM EMPLOYEES
WHERE DEPARTMENT_ID IN (10,20,30,40) GROUP BY
ROLLUP(DEPARTMENT_ID);
```

ORDER BY

This clause is used to specify in which order the result should be displayed either ascending (this is default) or descending order. You can use one or more columns, functions and column position (which should be greater than 0) in order by clause. Whenever you use this clause then 'SORT' operation takes place. Null comes last in ascending order and first in descending order.

Rules-

If you use distinct or group by clause in select statement then order by column must be in select list.

If you are using sub-query order by must be placed at the end of sub query.

Order by clause prevents a SELECT statement from being an updatable cursor.

Example

```
SELECT COMMISSION_PCT FROM EMPLOYEES ORDER BY
COMMISSION_PCT,SALARY;
SELECT COMMISSION_PCT+SALARY FROM EMPLOYEES ORDER BY
COMMISSION_PCT+SALARY;
```

```
SELECT COMMISSION_PCT FROM EMPLOYEES ORDER BY SALARY DESC;
SELECT DISTINCT(COMMISSION_PCT) FROM EMPLOYEES ORDER BY SALARY
DESC;
SELECT FIRST_NAME, COUNT(*) FROM EMPLOYEES GROUP BY FIRST_NAME
ORDER BY COUNT(*);
SELECT COMMISSION_PCT FROM EMPLOYEES ORDER BY 1;
SELECT SALARY FROM EMPLOYEES ORDER BY F2(SALARY);
```

WITH

It is processed as temporary table and it is used to simplify complex queries.

Example

```
WITH DEPT_COUNT AS (SELECT DEPARTMENT_ID, COUNT(*) AS DEPT_COUNT
FROM EMPLOYEES GROUP BY DEPARTMENT_ID)
SELECT E.FIRST_NAME, D.DEPT_COUNT
FROM EMPLOYEES E, DEPT_COUNT D
WHERE E.DEPARTMENT_ID = D.DEPARTMENT_ID
```

PL SQL:

PL/SQL is a procedural language extension to SQL with designed features of programming language. Users can declare variables, cursors and handle errors.PL/SQL groups logically related SQL statements put together in one single block and send the entire block to the server in one single call. There by reducing network traffic.

Example:

```
BEGIN
DBMS_OUTPUT.PUT_LINE('HELLO WORLD');
END;
/
```

Output:

HELLO WORLD

Difference between PL/SQL and SQL:

1) SQL is executed one statement at a time. PL/SQL is executed as a block of code.
2) You can embed SQL in a PL/SQL program but you cannot embed PL/SQL in a SQL statement.
3) SQL tells the database what to do but PL/SQL tell the database how to do things.

The attribute of variable are:

1) %TYPE

This one is used to retrieve data type of single column from a table.

2) %ROWTYPE

This attribute is used to retrieve data type of entire column from table. Here you don't have declared variables for all the columns in a table and you don't have to update code when you alter the column in a table. When you use this the memory will be used for creating data type for all the columns, so use this only when you are selecting all columns.

3) RECORD

Records are composite data type which is combination of different scalar data type like char, varchar, number…etc. This attribute is used to retrieve entire row or specified columns from table.

Example:

```
TYPE GREENS IS RECORD(A VARCHAR(20), B NUMBER(20), C DATE);
```

4) PL/SQL TABLE

It is an ordered collection of element of the same data type. Each element has unique subscript number which can identify the position. PL/SQL table is unbounded. It has some attributes they are

EXISTS (n) – Return true if the nth element is present.
COUNT – It is used to get total number of elements in PL/SQL table
FIRST – It is used to get first index or subscript value.
LAST - It is used to get last index or subscript value.
PRIOR(n) – It returns the prior index number of n.
NEXT(n) – It returns the next index number of n.
DELETE – It is used to delete the all elements (DELETE), particular elements (DELETE (n)) or ranging elements from & to (DELETE (m.n)).

Example:
TYPE GREENS IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
I GREENS;

5) VARRAY

Varray stands for variable-size array. When you create varray you must declare maximum number of columns. It is similar to PL/SQL table and only difference from PL/SQL table is that you must mentions no of columns.

Example:
TYPE TNAME IS VARRAY(3) OF VARCHAR2(20);
I TNAME;

EXCEPTIONS:

Exceptions are error handling action to be performed when error occurs.
Types of exception are

1) Pre-Defined

This can be called as system defined exception or named system exception. PL/SQL predefines some common oracle errors as exceptions. There is 20 pre-defined exceptions. The list of pre define exception are

SYS_INVALID_ROWID -Exception occurs when row-id not found.

CURSOR_ALREADY OPENED-Exception occurs when you open a cursor which is already opened.
NO_DATA_FOUND -Exception occurs when a SELECT…INTO clause doesn't return any row from a table.
TOO_MANY_ROWS -Exception occurs when SELECT more than one row in a variable or record.
ZERO_DIVIDE -Whenever we divide any number by zero the output would be 'ORA-01476 divisor is equal to zero'. Exception occurs when you try to divide a number by zero.
INVALID_NUMBER- Exception occurs when you convert string to number.
VALUE_ERROR - Exception occurs when conversion fails (i.e. varchar to number) and if the value is longer than the declared length of the variable.
OTHERS - This is used to trap all remaining exceptions that has not handled by above four predefined exception and can handle above exception too. This exception must be last when you have more than one predefined exceptions.

2) User-Defined

Apart from system exception you could explicitly define and raise exception. These are known as user-defined exception. In user-defined we use 'RAISE' keyword to forcibly raise an exception.
Rules to be followed

1) Exception must be declared at the declaration section.
2) Exception should be raised in execution section.
3) Exception should be handled in exception section.

3) Non-Pre-Defined

The system exception for which oracle doesn't have names is called non pre-defined or unnamed system exception. Here we use 'PRAGMA EXCEPTION_INIT' is used to bind user defined exception to a particular error number or 'OTHERS'.

RAISE_APPLICATION_ERROR

This is used to display user defined error message with error umber ranging from -20000 to -20999.

Rules for writing exception

1) If you use more than one exception in a block then only one exception fires that will be based on first error occurrence.
2) If you use 'OTHERS' exception with another exception then 'OTHERS' must be last in exception section.

CURSORS

Cursor is a SQL private work area. It opens an area of memory where the query is parsed and executed. It stores the result of the query and manipulates in temporary work area. A cursor can hold more than 1 rows, but it process row by row. The set of rows the cursor holds is called the active set.

Cursor attributes:

1) FOUND – Returns true if an INSERT, DELETE & UPDATE affect one or more rows or a SELECT INTO returns one or more rows. Otherwise it returns false.

2) NOTFOUND – It is opposite to 'FOUND'. Returns true if an INSER, DELETE & UPDATE affected no rows or a SELECT INTO returned no rows. Otherwise it returns false.

3) ROWCOUNT – Returns number of rows affected by INSERT, DELETE & UPDATE or returned by SELECT INTO statements.

4) IS OPEN – Returns true if cursor is opened and it is false hen cursor is not opened.

Types of cursor:

1) Implicit Cursor

They are created automatically when DML statements like INSERT, UPDATE and DELETE statements are executed and SELECT statement that return just one row is executed.

Example

```
BEGIN
DELETE FROM T1;
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
INSERT INTO T1 VALUES(1);
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
END;
/
```

Output:-

```
1
1
```

Example

```
BEGIN
DELETE FROM T1 WHERE A=2;
IF SQL%FOUND THEN DBMS_OUTPUT.PUT_LINE('DELETE SUCCEEDED');
INSERT INTO T1 VALUES(4);
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
END IF;
END;
/
```

Output:-

```
DELETE SUCCEEDED
1
```

Example

```
BEGIN
DELETE FROM T1 WHERE A=5;
```

```
IF SQL%NOTFOUND THEN DBMS_OUTPUT.PUT_LINE('DELETE UNSUCESSFUL');
INSERT INTO T1 VALUES(4);
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT);
END IF;
END;
/
```

Output:-

```
DELETE UNSUCESSFUL
1
```

2) Explicit Cursor

These should be declared explicitly by user when 'SELECT' keyword returns more than 1 rows or no rows. Though it stores multiple rows, only one row can be processed at a time which is called current row.

Example

```
DECLARE
I EMPLOYEES.FIRST_NAME%TYPE;
J EMPLOYEES.SALARY%TYPE;
CURSOR C1 IS SELECT FIRST_NAME,SALARY FROM EMPLOYEES WHERE
DEPARTMENT_ID=60;
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO I,J;
DBMS_OUTPUT.PUT_LINE(I||' '||J);
EXIT WHEN C1%FOUND;
END LOOP;
CLOSE C1;
END;
/
```

Output:-
Alexander 9000

SELECT for UPDATE

This is used to lock the rows in cursor result set until the transaction completes and other transactions are blocked from updating or deleting these rows. Change to the record is not mandatory when you use this statement. These lock will be released when you commit or rollback the transaction. NOWAIT is optional keyword. This is used to lock row when rows are locked by another user, the control is immediately returned to your program. If you omit this keyword then it waits until the row is available.

Example

```
DECLARE
CURSOR C1 IS SELECT A FROM T1 FOR UPDATE OF A NOWAIT;
BEGIN
OPEN C1;
UPDATE T1 SET A=5 WHERE A=6;
CLOSE C1;
COMMIT;
END;
/
```

Output:-
PL/SQL procedure successfully completed.

WHERE CURRENT OF

This is used when you update or delete rows using cursor. This allows you to update or delete the row currently being addressed, without creating reference to ROWID. When you use this then the cursor must contain FOR UPDATE clause.

Example

```
DECLARE
I T1.A%TYPE;
CURSOR C1 IS SELECT A FROM T1 WHERE A=4 FOR UPDATE OF A;
BEGIN
OPEN C1;
FETCH C1 INTO I;
IF C1%NOTFOUND THEN
DBMS_OUTPUT.PUT_LINE('NO ROWS FOUND');
ELSE
UPDATE T1 SET A=9 WHERE CURRENT OF C1;
COMMIT;
END IF;
CLOSE C1;
END;
/
```

Output:-
PL/SQL procedure successfully completed.

Example

```
DECLARE
I T1.A%TYPE;
CURSOR C1 IS SELECT A FROM T1 WHERE A=9 FOR UPDATE OF A;
BEGIN
OPEN C1;
FETCH C1 INTO I;
IF C1%NOTFOUND THEN
DBMS_OUTPUT.PUT_LINE('DATA NOT FOUND');
ELSE
DELETE FROM T1 WHERE CURRENT OF C1;
END IF;
CLOSE C1;
END;
/
```

Output:-
PL/SQL procedure successfully completed.

CURSOR WITH PARAMETER

Example

```
DECLARE
A EMPLOYEES.FIRST_NAME%TYPE;
CURSOR EMP_CURSOR
(A EMPLOYEES.DEPARTMENT_ID%TYPE) IS
SELECT FIRST_NAME FROM EMPLOYEES WHERE DEPARTMENT_ID=A;
BEGIN
OPEN EMP_CURSOR(60);
LOOP
FETCH EMP_CURSOR INTO A;
DBMS_OUTPUT.PUT_LINE(A);
EXIT WHEN EMP_CURSOR%NOTFOUND;
END LOOP;
CLOSE EMP_CURSOR;
OPEN EMP_CURSOR(20);
LOOP
FETCH EMP_CURSOR INTO A;
DBMS_OUTPUT.PUT_LINE(A);
EXIT WHEN EMP_CURSOR%NOTFOUND;
END LOOP;
CLOSE EMP_CURSOR;
END;
/
```

Output:-
Alexander
Bruce
David
Valli

Diana
Diana
Michael
Pat
Pat

When to use cursor

1) If you want to process row by row then use cursor.

Disadvantage of cursor

1) Performance drops down.

REF CURSOR:

It is a data type. It will close implicitly. There are two types of ref cursor

a) Weak ref cursor

Query result will be stored.

Example

```
CREATE OR REPLACE PROCEDURE P1( I IN NUMBER, O OUT SYS_REFCURSOR)
AS
BEGIN
OPEN O FOR
SELECT FIRST_NAME,SALARY FROM EMPLOYEES WHERE EMPLOYEE_ID=I;
END;
/

VARIABLE O REFCURSOR;

EXEC P1(90,:O);
```

b) Strong ref-cursor

It returns a value which can be of any data type.

Example

```
CREATE OR REPLACE PACKAGE PKG1 AS
TYPE TNAME IS REF CURSOR RETURN DEPARTMENTS%ROWTYPE;
END PKG1;
/
```

STORED PROCEDURE

It is a named PL/SQL sub programs. It is compiled and stored in database for repeated execution. It can accept an argument and can return a value. Procedure performs an action. It is a schema object. It improves the performance of an application by reducing network traffic. It can accept parameter. If procedure is created outside the package then it is called as stored or standalone subprograms and if is created in package then it is called packaged subprogram.

- We can create procedure inside anonymous block.
- We can create procedure inside package.
- We can create procedure inside procedure.

Example

```
CREATE PROCEDURE P1 AS
K EMPLOYEES.FIRST_NAME%TYPE;
L EMPLOYEES.SALARY%TYPE;
BEGIN
SELECT FIRST_NAME, SALARY
INTO K,L
FROM EMPLOYEES WHERE EMPLOYEE_ID=100;
DBMS_OUTPUT.PUT_LINE(K||' '||L);
END;
/
```

To alter procedure:

Example

```
CREATE OR REPLACE PROCEDURE P1 AS
K EMPLOYEES.FIRST_NAME%TYPE;
L EMPLOYEES.SALARY%TYPE;
BEGIN
SELECT FIRST_NAME, SALARY
INTO K,L
FROM EMPLOYEES WHERE EMPLOYEE_ID=100;
DBMS_OUTPUT.PUT_LINE(K||' '||L);
END;
/
```

To execute procedure:

SQLPLUS

```
EXEC P1;
```

iSQLPLUS

```
BEGIN
P1;
END;
```

To drop procedure:

```
DROP PROCEDURE P1;
```

To describe procedure:

Example

```
DESCRIBE P2;
```

To view entire query of an stored procedure

Example
SELECT TEXT FROM USER_SOURCE WHERE NAME='P2';

⇨ Is a positional parameter.

Local Procedure:

Whenever you create procedure inside anonymous block is called local procedure. This procedure doesn't store in database and cannot be called outside anonymous block.

Example

```
DECLARE
PROCEDURE P2
AS
A EMPLOYEES.FIRST_NAME%TYPE;
BEGIN
SELECT FIRST_NAME
INTO A
FROM EMPLOYEES WHERE EMPLOYEE_ID = 100;
DBMS_OUTPUT.PUT_LINE(A);
END;
BEGIN
INSERT INTO T1 VALUES(3,'SHELL');
COMMIT;
P2;
INSERT INTO T1 VALUES(1,'ORACLE');
ROLLBACK;
END;
/
```

Parameter types:

IN:

This indicates that values must be supplied when you execute the procedure.

Example

```
CREATE OR REPLACE PROCEDURE P2 ( A IN NUMBER)
AS
B EMPLOYEES.FIRST_NAME%TYPE;
BEGIN
SELECT FIRST_NAME
INTO B
FROM EMPLOYEES WHERE EMPLOYEE_ID=A;
DBMS_OUTPUT.PUT_LINE(B);
END;
/
```

Output
Steven

OUT:

Example

```
VARIABLE B NUMBER;
CREATE OR REPLACE PROCEDURE P1 (A IN NUMBER,B OUT NUMBER)
AS
BEGIN
DBMS_OUTPUT.PUT_LINE(A);
END;
/
EXEC P1(10,:B);
```

Output
10

IN/OUT:

Example

```
VARIABLE B NUMBER;

EXEC :B := 10;

CREATE OR REPLACE PROCEDURE P1 (C IN OUT NUMBER)
AS
BEGIN
DBMS_OUTPUT.PUT_LINE(C);
END;
/

EXEC P1(:B);
```

Output
10

FUNCTIONS :

It is a named PL/SQL program which can be called as stored function. It is compiled and stored in database for repeated execution.
It can accept arguments and must return a value. You can have multiple return statements but only one will be executed. We can create function at client-side application using procedure builder.
Function computes a value.
Function can be called as part of an execution.
Function cannot perform DML.
Function can return only one value.
Function can use only parameter IN

It is a schema object

- We can create function inside package.
- We can create function inside anonymous block.

- We can create function inside procedure.

If function is created outside the package then it is called as stored or standalone sub programs and if it created inside package then it is called packaged sub programs.

Example

To create functions

```
CREATE FUNCTION F1(A IN NUMBER)
RETURN VARCHAR2
AS
B EMPLOYEES.FIRST_NAME%TYPE;
BEGIN
SELECT FIRST_NAME
INTO B
FROM EMPLOYEES WHERE EMPLOYEE_ID=100;
DBMS_OUTPUT.PUT_LINE(B);
RETURN B;
END;
/
```

To alter function

```
CREATE OR REPLACE FUNCTION F1(A IN NUMBER)
RETURN VARCHAR2
AS
B EMPLOYEES.FIRST_NAME%TYPE;
BEGIN
SELECT FIRST_NAME
INTO B
FROM EMPLOYEES WHERE EMPLOYEE_ID=100;
DBMS_OUTPUT.PUT_LINE(B);
RETURN B;
END;
/
```

To drop function

DROP FUNCTION F1;
To execute a function
SELECT F1(100) FROM DUAL;
To view columns in function
DESCRIBE F1;
To view entire coding of an function
SELECT TEXT FROM USER_SOURCE WHERE NAME='F1';
Parameter
IN
Example

```
CREATE OR REPLACE FUNCTION F1(A IN NUMBER)
RETURN VARCHAR2
AS
B EMPLOYEES.FIRST_NAME%TYPE;
BEGIN
SELECT FIRST_NAME
INTO B
FROM EMPLOYEES WHERE EMPLOYEE_ID=100;
DBMS_OUTPUT.PUT_LINE(B);
RETURN B;
END;
/
```

Output
steven

Recursive function
When a subprogram calls itself is refereed as recursive call and the process is known as recursion.

Example

```
DECLARE
```

```
FIRST VARCHAR2(40);
FUNCTION F1(A NUMBER)
RETURN VARCHAR2
AS
B EMPLOYEES.FIRST_NAME%TYPE;
BEGIN
SELECT FIRST_NAME
INTO B
FROM EMPLOYEES WHERE EMPLOYEE_ID=100;
DBMS_OUTPUT.PUT_LINE(B);
RETURN B;
END F1;

BEGIN
FIRST := F1(10);
DBMS_OUTPUT.PUT_LINE('NAME '||FIRST);
END;
/
```

Output

```
steven
NAME steven
```

Invoking function in SQL Statement

Example

```
CREATE OR REPLACE FUNCTION F2(P_VALUE IN NUMBER)
RETURN NUMBER IS
BEGIN
RETURN (P_VALUE * 0.08);
END F2;
/
```

Output:

```
SELECT FIRST_NAME, LAST_NAME, SALARY, F2(SALARY)
FROM
```

EMPLOYEES WHERE DEPARTMENT_ID=100;

FIRST_NAME	LAST_NAME	SALARY	F2(SALARY)
Nancy	Greenberg	12000	960
Daniel	Faviet	9000	720
John	Chen	8200	656
Ismael	Sciarra	7700	616

Difference between Stored Procedure and Fuctions:

1.) Procedure perform an action and Functions perform compute values.
2.) Procedure can accept and can return a value. But, functions can accept a value and must return a value.
3.) Functions can be call from SQL Statements, procedures not.
4.) Functions are generally restricted to returning a single value, while procedures can have multiple OUT parameters.
5.) There are situations where functions are more useful in queries (i.e. you want to create your own function that can be used in a query like the standard functions UPPER, etc.). In some client-side languages, it's slightly easier to call stored procedures rather than stored functions.

TRIGGER:

Trigger is a names PL/SQL block that is stored in database and invoked repeatedly. It automatically fires when an event occurs in schema, tables or database. You can enable and disable trigger.

Trigger is used for

- Prevent invalid transaction
- Table modification
- Complex business rule.

Restrictions

- We cannot use COMMIT, SAVEPOINT & ROLLBACK in trigger.

- We cannot use DDL in trigger because they have implicit commit.

Example

To create trigger:

```
CREATE TRIGGER TRG_T1
BEFORE INSERT OR UPDATE OR DELETE ON T1
FOR EACH ROW
BEGIN
INSERT INTO T2 VALUES(1);
END;
/
```

To update Trigger:

```
CREATE OR REPLACE TRIGGER TRG_T1
BEFORE INSERT OR UPDATE OR DELETE ON T1
BEGIN
INSERT INTO T2 VALUES(1);
END;
/
```

To drop trigger:

```
DROP TRIGGER TRG_T1;
```

To enable a particular trigger:

```
ALTER TRIGGER TRG_T1 ENABLE;
```

To enable all triggers for an table:

```
ALTER TABLE T1 ENABLE ALL TRIGGERS;
```

To disable a particular trigger:

```
ALTER TRIGGER TRG_T1 DISABLE;
```

To disable all triggers for an table:

```
ALTER TABLE T1 DISABLE ALL TRIGGERS;
```

To view trigger query:

```
SELECT TEXT FROM USER_SOURCE WHERE NAME='TRG_T1';
```

Trigger Types:

1.) Statement Level Trigger (or) Table Level Trigger

It fires only once for each event. This is a default trigger. It fires once even if no rows are affected by the triggering event.

Example
Before Trigger:

```
CREATE OR REPLACE TRIGGER TRG_T1
BEFORE INSERT OR UPDATE OR DELETE ON T1
BEGIN
INSERT INTO T2 VALUES(1);
END;
```

After Trigger:

```
CREATE OR REPLACE TRIGGER TRG_T1
AFTER INSERT OR UPDATE OR DELETE ON T1
BEGIN
INSERT INTO T2 VALUES(1);
END;
/
```

Instead Of:

```
CREATE TRIGGER TRG_T1
INSTEAD OF INSERT OR UPDATE OR DELETE ON V1
BEGIN
INSERT INTO T2 VALUES(1);
END;
/
```

2.) Row Level Trigger:

It fires once for each row affected by the triggering event. This trigger will not fire when no rows are affected by triggering event.

Example
Before Trigger:

```
CREATE TRIGGER TRG_T1
BEFORE INSERT OR UPDATE OR DELETE ON T1
FOR EACH ROW
BEGIN
INSERT INTO T2 VALUES(1);
END;
/
```

After Trigger:

```
CREATE TRIGGER TRG_T1
AFTER INSERT OR UPDATE OR DELETE ON T1
FOR EACH ROW
BEGIN
INSERT INTO T2 VALUES(1);
END;
/
```

Instead Of:

```
CREATE TRIGGER TRG_T1
INSTEAD OF INSERT OR UPDATE OR DELETE ON V1
FOR EACH ROW
BEGIN
INSERT INTO T2 VALUES(1);
END;
/
```

1) DML Triggers

DML Triggers is created on either table or view and triggering event composing of Insert, Update & Delete. To create trigger on merge statement, create trigger on Insert & update statement to which the merge operation decomposes.

2) System Trigger

If a trigger is created on Schema or database then the triggering even is composed of data definition language or database operations.

a) Schema triggers

It includes DDL statements CREATE, ALTER and DROP.

Example

```
CREATE OR REPLACE TRIGGER TRG_T1
BEFORE DROP ON HR.SCHEMA
BEGIN
INSERT INTO T1 VALUES(1);
END;
/
```

b) Instead of Create Trigger

Example

```
CREATE TRIGGER TRG_T1
INSTEAD OF CREATE ON SCHEMA
BEGIN
```

```
INSERT INTO T1 VALUES(1);
END;
/
```

c) Database Trigger

It includes database startup and shutdown.

Example

```
CREATE TRIGGER TRG_T1
AFTER LOGON ON DATABASE
BEGIN
INSERT INTO T2 VALUES(1);
END;
/

CREATE TRIGGER TRG_T2
BEFORE LOGOFF ON DATABASE
BEGIN
INSERT INTO T2 VALUES(2);
END;
/
```

3) Conditional Triggers

In this we use 'WHEN' clause which specifies an SQL condition and that evaluated for each row. When you use 'WHEN' clause then you must definitely use 'FOR EACH ROW'.

Trigger Timings:

1) BEFORE – Trigger fires first and then event fires.
2) AFTER – Event fires first and then triggers fires.
3) INSTEAD OF – this is use when you create trigger on view. This stops inserting, deleting and updating a view.

Trigger Events:

Data Manipulation Language (Insert, Update & Delete)
Data Definition Language (Drop, Create, Truncate, Rename)
Logon, Logoff

PACKAGES:

Package groups logically related sub-programs (procedure and function is called subprogram). It is an schema object. It consists of 2 parts

1) Package specification

It contains declaration. You can declare cursor, type, variables, sub programs, exceptions and constant. You can create package specification with package body. Whatever declared in package specification is global.

2) Package body

It contains definition. It contains the definition for subprograms that is listed in package specification and it contains local procedure and one time procedure.
You can delete package body without deleting package specification.
It cannot be invoked, parameterized or nested.

Advantage:

1. It allow oracle server to read multiple object into memory at once.
2. Overloading – multiple sub programs at same time.
3. Users can declare global variables, cursors & user-defined exception.
4. We can use 'PRAGMA SERIALLY_REUSABLE' to de-allocate memory after each call. If you are using this you need to use both in package body & package specification before definition or declaration.

Example

```
CREATE OR REPLACE PACKAGE PKG2 AS
--DECLARTION
--GLOBAL VARIABLE
OTP VARCHAR2(50);
FNAME EMPLOYEES.FIRST_NAME%TYPE;
EID EMPLOYEES.EMPLOYEE_ID%TYPE;
SAL EMPLOYEES.SALARY%TYPE;
--GLOBAL VARIABLE
```

```
--SUBPROGRAMS
PROCEDURE P1(EID IN EMPLOYEES.EMPLOYEE_ID%TYPE);
PROCEDURE P2(FNAME IN EMPLOYEES.FIRST_NAME%TYPE, LNAME IN
EMPLOYEES.LAST_NAME%TYPE);
PROCEDURE P2(FNAMEE IN EMPLOYEES.FIRST_NAME%TYPE, LNAMEE IN
EMPLOYEES.LAST_NAME%TYPE);
PROCEDURE P2(FNAME IN EMPLOYEES.FIRST_NAME%TYPE, LNAME IN
EMPLOYEES.LAST_NAME%TYPE, EID IN EMPLOYEES.EMPLOYEE_ID%TYPE);
FUNCTION F1(EID IN EMPLOYEES.EMPLOYEE_ID%TYPE) RETURN NUMBER;
FUNCTION TAX(TAXRATE IN NUMBER) RETURN NUMBER;
PROCEDURE AWR_BONUS(A IN NUMBER, B IN NUMBER);
--SUBPROGRAMS
--DECLARTION
END PKG2;
/


CREATE OR REPLACE PACKAGE BODY PKG2 AS
--DEFINITION
--LOCAL PROCEDURE
PROCEDURE P3(A IN NUMBER, B IN NUMBER, C IN NUMBER)
AS
BEGIN
DBMS_OUTPUT.PUT_LINE(A+B+C);
END P3;
--LOCAL PROCEDURE
--USER DEFINED PACKAGE
FUNCTION TAX(TAXRATE IN NUMBER)
RETURN NUMBER
AS
TAXPER NUMBER := 0.08;
BEGIN
RETURN (TAXRATE*TAXPER);
END TAX;
--USER DEFINED PACKAGE
PROCEDURE P1(EID IN EMPLOYEES.EMPLOYEE_ID%TYPE)
```

```
AS
BEGIN
SELECT FIRST_NAME INTO FNAME FROM EMPLOYEES WHERE
EMPLOYEE_ID=EID;
DBMS_OUTPUT.PUT_LINE(FNAME);
END P1;
PROCEDURE P2(FNAME IN EMPLOYEES.FIRST_NAME%TYPE, LNAME IN
EMPLOYEES.LAST_NAME%TYPE)
AS
BEGIN
SELECT EMPLOYEE_ID INTO EID FROM EMPLOYEES WHERE
FIRST_NAME=FNAME AND LAST_NAME=LNAME;
P3(2,3,5);
DBMS_OUTPUT.PUT_LINE('1' || ' ' || FNAME || ' - ' || LNAME || ' - ' || EID);
END P2;
PROCEDURE P2(FNAMEE IN EMPLOYEES.FIRST_NAME%TYPE, LNAMEE IN
EMPLOYEES.LAST_NAME%TYPE)
AS
BEGIN
SELECT EMPLOYEE_ID INTO EID FROM EMPLOYEES WHERE
FIRST_NAME=FNAMEE AND LAST_NAME=LNAMEE;
DBMS_OUTPUT.PUT_LINE('2' || ' ' ||FNAMEE || ' - ' || LNAMEE || ' - ' || EID);
END P2;
PROCEDURE P2(FNAME IN EMPLOYEES.FIRST_NAME%TYPE, LNAME IN
EMPLOYEES.LAST_NAME%TYPE, EID IN EMPLOYEES.EMPLOYEE_ID%TYPE)
AS
BEGIN
SELECT SALARY INTO SAL FROM EMPLOYEES WHERE FIRST_NAME=FNAME
AND LAST_NAME=LNAME AND EMPLOYEE_ID=EID;
DBMS_OUTPUT.PUT_LINE(FNAME || ' - ' || LNAME || ' - ' || SAL);
END P2;
FUNCTION F1(EID IN EMPLOYEES.EMPLOYEE_ID%TYPE)
RETURN NUMBER
AS
SAL EMPLOYEES.SALARY%TYPE;
BEGIN
```

```
SELECT SALARY INTO SAL FROM EMPLOYEES WHERE EMPLOYEE_ID=EID;
RETURN SAL;
END F1;
--FORWARD DECLARTAION
PROCEDURE CAL_RATE(A IN NUMBER, B IN NUMBER);
--FORWARD DECLARTAION
PROCEDURE AWR_BONUS(A IN NUMBER, B IN NUMBER)
IS
BEGIN
CAL_RATE(A,B);
END AWR_BONUS;
PROCEDURE CAL_RATE(A IN NUMBER, B IN NUMBER)
IS
BEGIN
DBMS_OUTPUT.PUT_LINE(A+B);
END CAL_RATE;
--ONE TIME ONLY PROCEDURE
BEGIN
SELECT FIRST_NAME INTO OTP FROM EMPLOYEES WHERE EMPLOYEE_ID=OTP;
DBMS_OUTPUT.PUT_LINE(OTP);
--ONE TIME ONLY PROCEDURE
--DEFINITION
END PKG2;
/
```

Example

To execute procedure from package

```
EXEC PKG2.P2('Steven','King',100);
```

To execute function from package

```
SELECT PKG2.F1(100) FROM DUAL;
```

To execute user defined package

```
SELECT PKG2.TAX(SALARY), SALARY FROM EMPLOYEES;
```

To drop package:

```
DROP PACKAGE PKG3;
```

To drop package body:
DROP PACKAGE BODY PKG3;

One time only procedure:

It is executed only once in the current session. You can't use 'END' in one time only procedure. This should be defined last in package body.
Example
To execute one time only procedure

```
DECLARE
NAME VARCHAR2(50);
BEGIN
NAME := PKG2.OTP;
DBMS_OUTPUT.PUT_LINE(NAME);
END;
/
```

Bodiless package:
You can use bodiless package for execution. Here these variables exist only in this session.

Example

```
CREATE OR REPLACE PACKAGE PKG3 AS
NUM1 CONSTANT NUMBER := 10;
NUM2 CONSTANT NUMBER := 11;
NUM3 CONSTANT NUMBER := 12;
NUM4 CONSTANT NUMBER := 13;
END PKG3;
/
EXECUTE DBMS_OUTPUT.PUT_LINE('RESULT'||'='||20*PKG3.NUM3);
```

Output
240

Forward Declaration:
When the sub programs are defined in alphabetical order and you cannot call procedure CAL_RATE' from 'AWR_BONUS'.

So before calling procedure ‘CAL_RATE’ we must define it.

PRAGMA

It refers to compiler directive which is used to provide information to the compiler.

PRAGMA AUTONOMIOUS_TRANSACTION

It performs independent transaction between begin and end without affecting entire transaction or other transaction.

You must use ‘COMMIT’ or ‘ROLLBACK’ else we get an error ‘ORA-06519 active autonomous transaction detected and rollback.

Example

```
DECLARE
PROCEDURE P1
AS
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
INSERT INTO T1 VALUES(2);
COMMIT;
END;
BEGIN
INSERT INTO T1 VALUES(1);
P1;
INSERT INTO T1 VALUES(3);
ROLLBACK;
END;
/
```

PRAGMA EXCEPTION_INIT:

It is used in exception. It is used to handle the error for which we know the error number is called non-pre-defined exception.

Example

```
DECLARE
GREENS EXCEPTION;
```

```
PRAGMA EXCEPTION_INIT(GREENS,-01476);
BEGIN
DBMS_OUTPUT.PUT_LINE(1/0);
DBMS_OUTPUT.PUT_LINE('PRINT 1');
EXCEPTION
WHEN GREENS THEN
DBMS_OUTPUT.PUT_LINE('PRINT 2');
END;
/
```

Output:-

PRINT 2

PRAGMA SERIALLY_REUSABLE:

It is used to use package for single call to the server after the call is made then package is released from memory to reclaim in another call to server.

Example

```
CREATE OR REPLACE PACKAGE PKG1 IS
PRAGMA SERIALLY_REUSABLE;
NUM NUMBER := 0;
PROCEDURE ENTER(N NUMBER);
PROCEDURE SHOW;
END PKG1;
/
CREATE OR REPLACE PACKAGE BODY PKG1 IS
PRAGMA SERIALLY_REUSABLE;
PROCEDURE ENTER (N NUMBER) IS
BEGIN
PKG1.NUM := N;
END;
PROCEDURE SHOW IS
BEGIN
DBMS_OUTPUT.PUT_LINE('Num: ' || PKG1.NUM);
END;
END PKG1;
/
```

EXECUTE IMMEDIATE

This command is used for executing dynamic SQL statements and data definition language commands.

Example

```
BEGIN
EXECUTE IMMEDIATE 'TRUNCATE TABLE T1';
END;
/
```

Dynamic SQL:

Dynamic SQL is a SQL statement that is constructed and executed at program execution time.

Example

```
DECLARE
I VARCHAR2(20) := 'B1';
J NUMBER;
BEGIN
EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || I INTO J;
DBMS_OUTPUT.PUT_LINE(J);
END;
/
```

Output

3

DBMS_SQL :

It is similar to ‘EXECUTE IMMEDIATE’ to run dynamic SQL and DML statements. If this is running in anonymous block then it checks for privilege of the current user and if it is running in stored procedure then it checks for owner of the stored procedure.

Example

```
CREATE OR REPLACE PROCEDURE DELETE_ALL (NAME IN VARCHAR2,
ROWS_DELETE OUT NUMBER)
IS
```

```
CURSOR_NAME INTEGER;
BEGIN
CURSOR_NAME := DBMS_SQL.OPEN_CURSOR;
DBMS_SQL.PARSE(CURSOR_NAME,'DROP TABLE
'||NAME,DBMS_SQL.NATIVE);
ROWS_DELETE := DBMS_SQL.EXECUTE(CURSOR_NAME);
DBMS_SQL.CLOSE_CURSOR(CURSOR_NAME);
END;
/

VARIABLE DELETED NUMBER
EXEC DELETE_ALL('STUDENT',:DELETED);
PRINT DELETED;
```

DBMS_JOBS

It is a schedule PL/SQL program to run. It is very useful in running batch jobs during no peak hours or to run maintenance during times of low usage.

Example

```
VARIABLE J NUMBER
BEGIN
DBMS_JOB.SUBMIT(
JOB => :J,
WHAT => 'P1;',
NEXT_DATE => TRUNC(SYSDATE),
INTERVAL => 'SYSDATE+1/1440'
);
COMMIT;
END;
/
```

Parameter in Jobs

JOB – Uniquely identifies the job or job name.
WHAT – PL/SQL code to execute as a job.
NEXT_DATE – Next execution date of the job.
INTERVAL – Specifies in which interval it should run.

To enable jobs

```
BEGIN
DBMS_JOB.BROKEN(121,FALSE);
END;
```

To disable jobs

```
BEGIN
DBMS_JOB.BROKEN(121,TRUE);
END;
```

To force jobs

```
BEGIN
DBMS_JOB.RUN(122);
END;
```

To remove jobs

```
BEGIN
DBMS_JOB.REMOVE(121);
END;
```

To list out the jobs present in the database

```
SELECT * FROM USER_JOBS;
```

DBMS_DDL

This is used to access some SQL DDL statements from stored procedure. This package allows developer to access to ALTER and ANALYZE SQL statements through PL/SQL environment. You can recompile procedure, function, package, package body or trigger using DBMS_DDL.ALTER_COMPILE and you can analyze table, cluster or index using DBMS_DDL.ANALYZE_OBJECT.

DBMS_OUTPUT

It is used to display output from PL/SQL blocks.

PUT

It stores or appends the output from procedure in the output buffer.

Example

```
DECLARE
A NUMBER;
BEGIN
A := 10;
DBMS_OUTPUT.PUT(A);
DBMS_OUTPUT.PUT_LINE(A);
END;
/
```

Output

```
1010
```

NEW_LINE

It helps to print the output in the new line.

Example

```
DECLARE
A NUMBER;
BEGIN
A := 10;
DBMS_OUTPUT.PUT(A);
DBMS_OUTPUT.NEW_LINE();
DBMS_OUTPUT.PUT_LINE(A);
END;
/
```

Output

```
10
10
```

PUT_LINE

It is a combination of PUT & NEW_LINE

Example

```
DECLARE
A NUMBER;
BEGIN
```

```
A := 10;
DBMS_OUTPUT.PUT_LINE(A);
END;
/
```

Output

10

GET_LINE

It consists of two arguments if there is no line in buffer then it is status will be 1 else 0.

Example

```
DECLARE
A NUMBER;
B NUMBER := 0;
BEGIN
A := 10;
DBMS_OUTPUT.PUT(A);
DBMS_OUTPUT.GET_LINE(A,B);
IF B = 0
THEN
DBMS_OUTPUT.NEW_LINE();
DBMS_OUTPUT.PUT_LINE(B);
ELSE
DBMS_OUTPUT.NEW_LINE();
DBMS_OUTPUT.PUT_LINE(B);
END IF;
END;
/
```

Output

10

1

GET_LINES

Example

Output

ENABLE/DISABLE

Example

```
DECLARE
A NUMBER;
BUFFER_SIZE NUMBER := 20000;
BEGIN
A := 10;
DBMS_OUTPUT.DISABLE;
DBMS_OUTPUT.PUT_LINE(A);
A := A+10;
DBMS_OUTPUT.ENABLE(BUFFER_SIZE);
DBMS_OUTPUT.PUT_LINE(A);
END;
/
```

Output

20

UTL_FILE

This package is used to access operating system files. With this package you can read from and write to operating system files. UTL_FILE consists of functions (FOPEN, IS_OPEN) & procedures (GET_LINE, PUT, PUT_LINE, PUTF, NEW_LINE, FFLUSH, FCLOSE, FCLOSE_ALL). It contains seven exception named as INAVLID_PATH, INVALID_MODE, INVALID_FILEHANDLE, INVALID_OPERATION, READ_ERROR, WRITE_ERROR and INTERNAL_ERROR.

UTL_HTTP

This package allows you to make HTTP request directly from database. It makes HTTP callouts from PL/SQL and SQL to access data on the internet. It contains two entry point functions: REQUEST and REQUEST_PIECES. The REQUEST function returns up to the first 2000 bytes of data from specified URL and REQUEST_PIECES returns a PL/SQL table of 2000 byte pieces if the data from specified URL. If an HTTP call fails (i.e. URL is not properly specified) then REQUEST_FAILED exception is raised and HTTP call fails for a lack of memory INIT_FAILED exception is raised. If there is no response from specified URL then the formatted HTML error message is returned.

Example

```
DECLARE
x utl_http.html_pieces;
BEGIN
x := utl_http.request_pieces('http://www.oracle.com/', 100);
dbms_output.put_line(x.count || ' pieces were retrieved.');
dbms_output.put_line('with total length ');
IF x.count < 1
THEN dbms_output.put_line('0');
ELSE dbms_output.put_line
 ((2000 * (x.count - 1)) + length(x(x.count)));
END IF;
END;
/
```

Output

```
17 pieces were retrieved.
with total length
33095
```

UTL_TCP

This package enables PL/SQL applications to communicate with external TCP/IP –based servers using TCP/IP. It has two functions namely OPEN_CONNECTION & CLOSE_CONNECTION. In OPEN_CONNECTION takes remote host and remote port to connect. CLOSE_CONNECTION takes previously opened connection to close. To close all connections which is opened use CLOSE_ALL_CONNECTIONS. READ function receives binary, text or line data from a service on an open connection. WRITE function transmits binary, text or line message to a service on an open connection. Exception are raised when buffer size is too small, network error occurs and bad arguments is passed in function call.

UTL_MAIL

The UTL_MAIL package provides a simple API to allow email to be sent from PL/SQL. In prior versions this was possible using the UTL_SMTP package (shown here), but this required knowledge of the SMTP protocol.
The UTL_MAIL utility is used to send e-mail and manage e-mail which includes commonly used e-mail features, such as attachments, CC, BCC… etc. It has SEND, SEND ATTACH RAW, SEND ATTACH VARCHAR2 procedures.

The package is loaded by running the following scripts.

```
CONN sys/password AS SYSDBA

@$ORACLE_HOME/rdbms/admin/utlmail.sql

@$ORACLE_HOME/rdbms/admin/prvtmail.plb

GRANT EXECUTE ON UTL_MAIL TO test_user;
```

In addition the SMTP_OUT_SERVER parameter must be set to identify the SMTP server.

```
CONN sys/password AS SYSDBA

ALTER SYSTEM SET smtp_out_server='smtp.domain.com' SCOPE=SPFILE;

SHUTDOWN IMMEDIATE

STARTUP
```

With the configuration complete we can now send a mail.

```
BEGIN
 EXECUTE IMMEDIATE 'ALTER SESSION SET smtp_out_server =
''192.168.152.1''';
  UTL_MAIL.send(sender => 'suresh_ji@yahoo.com',
          recipients => ' suresh_ji@yahoo.com ',
            subject => 'Test Mail',
            message => 'Hello World',
```

```
        mime_type => 'text; charset=us-ascii');
END;
/
```

PL/SQL COLLECTIONS:

Collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. It is one dimension.

Types of collections are

1) Associative array / PL/SQL Table

It can be called as index-by-table which is set of key-value pairs. Each key is unique and key can be either string or integer. It is capable of holding un-specified number of elements.

2) Nested table

Nested table is a column type that holds set of values. The rows are given consecutive subscripts starting at 1. It can be stored in a database.

3) Variable-size array / Records

V-array stands for variable-size array. When you create v-array you must declare maximum number of columns. They use sequential numbers as subscript.

SQL QUERY TUNNING

AWR Report

AWR can be abbreviated as automatic workload repository. It is used to collect performance statistics including session history, system statistics, time model statistics and object usage statistics.

TKPROF

It is an oracle utility to display trace file (.trc) in human readable format. A new trace file is created for every session and which contains session details.

EXPLAIN PLAN

It displays execution plan chosen by oracle for SELECT, INSERT, UPDATE and DELETE. Here we can see operation that has taken place, cost for the operation, no of rows affected, which object is used, how many bytes used ,

how much time taken and how it has been filtered. It let us know either which of these nested loops, hash join and sort merge are used or why it is been used. It shows you the cost taken to execute query based on this we try to optimize the query.

TYPE OF LOOPS

1. NESTED LOOPS

This operation is used when you join multiple tables using index.

2. HASH JOIN

This operation is used when you join multiple tables without index.

3. SORT MERGE

This operation is used when you join table other than equal to symbol (i.e. like >, <, >=, <=). Sort operation alone takes place when you use order by clause (i.e. without a join condition) & when you use aggregate functions (or) group functions and if you use both join & order by then sort merge operation takes place.

HINTS

Hints are used to force the query to run on particular index or full table scan to reduce cost. You must specify index name or full (table name) between /*+ */.

/*+ FULL(TABLE NAME) */ use this to run full table scan

INDEXES - speak on index –

EXTERNAL TABLE

External tables enable us to read flat-files (stored on the O/S) using SQL. They have been introduced in Oracle 9i as an alternative to SQL*Loader. External tables are essentially stored SQL*Loader control files, but because they are defined as tables, we can access our flat-file data using all available read-only SQL and PL/SQL operations. We can also read flat-files in parallel and join files to other files or tables, views and so on.

External table is similar to oracle table but you cannot perform DML operations and cannot create index on external table. It is used to move data in and out of the database. Example you are getting .csv from another department every day and you need to import into database. Instead of

writing command file every day create external table in which data will be dumped by running the script.

External tables are created using the SQL CREATE TABLE...ORGANIZATION EXTERNAL statement. When you create an external table, you specify the following attributes:

TYPE - specifies the type of external table. The two available types are the ORACLE_LOADER type and the ORACLE_DATAPUMP type. Each type of external table is supported by its own access driver.

1.) The ORACLE_LOADER access driver is the default. It can perform only data loads, and the data must come from text datafiles. Loads from external tables to internal tables are done by reading from the external tables' text-only datafiles.

2.) The ORACLE_DATAPUMP access driver can perform both loads and unloads. The data must come from binary dump files. Loads to internal tables from external tables are done by fetching from the binary dump files. Unloads from internal tables to external tables are done by populating the external tables' binary dump files.

DEFAULT DIRECTORY - specifies the default location of files that are read or written by external tables. The location is specified with a directory object, not a directory path. See Location of Datafiles and Output Files for more information.

ACCESS PARAMETERS - describe the external data source and implements the type of external table that was specified. Each type of external table has its own access driver that provides access parameters unique to that type of external table. See Access Parameters.

LOCATION - specifies the location of the external data. The location is specified as a list of directory objects and filenames. If the directory object is not specified, then the default directory object is used as the file location.

Step 1:

Create Flat-File:
Example: employees.dat

```
56-november, 15, 1980-  baker-          mary -     alice -09/01/2004
87-december, 20, 1970 - roper -         lisa  -    marie  -01/01/1999
56-november, 15, 1980-  baker-          mary -     alice -09/01/2004
87-december, 20, 1970 - roper -         lisa  -    marie  -01/01/1999
```

Step 2:

Directory Creation:

```
SQL> CREATE DIRECTORY xt_dir AS 'd:\oracle\dir';
Directory created.
```

Step 3:

Set Permission to Directory:

After a directory is created, the user creating the directory object needs to grant READ and WRITE privileges on the directory to other users.

```
GRANT READ ON DIRECTORY ext_tab_dir TO scott;
```

Step 4:

Create table emp_details

CREATE TABLE emp_details (emp_no CHAR(6), DOB date, first_name varCHAR2(20), middle_name varCHAR2(20), last_name varchar2(20), hire_date DATE);

Step 5:
Creating External Table name: exe_table_employee

```
create table exe_table_employee
(
        emp_id char(5),
        dob char(20),
        first_name char(20),
        middle_name char(20),
        last_name char(20),
        hire_date date
)
organization external
(
type oracle_loader
default directory ext_dir_suresh
access parameters
(
records delimited by newline
fields terminated by '-'
missing field values are null
(emp_id char(5),
        dob char(20),
        first_name char(20),
        middle_name char(20),
        last_name char(20),
        hire_date char(10) date_format date mask "mm/dd/yyyy"
        )
)
location ('info.dat')
);
```

Step 6:

Load the data from the external table exe_table_employee into the table emp_details:

```
SQL> INSERT INTO emp_details (emp_no, first_name,
middle_initial,last_name, hire_date, dob)
        (SELECT employee_number, employee_first_name,
substr(employee_middle_name, 1, 1),
        employee_last_name, employee_hire_date,
to_date(employee_dob,'month, dd, yyyy')
        FROM exe_table_employee);
```

Step 7:

```
Select * from emp_details;
Select * from exe_table_employee;
```

Step 8:

Load data internal table emp_details to external table import_employee.

```
create table exe_expo
organization external
(
type oracle_datapump
default directory ext_dir_sures
location ('inf.xls')
) as
select emp_id,
        dob,
        first_name,
        middle_name,
        last_name,
        hiredate
from suresh_emp;
```

SQL LOADER

SQL Loader is a bulk loader utility used for moving data from external files into the oracle database. It is used for high performance data loader

1) Load multiple data files into single table.
2) Load single data into multiple tables.

Open notepad and type this

```
CID,CNAME
ABC,XXX
10,ORACLE
20,UNIX
30,SHELL
```

Save the above file in .csv(comma separated value file) format.
Once again open notepad and type this

```
LOAD DATA INFILE 'C:\New\COURSE.CSV'
INTO TABLE COURSE
FIELDS TERMINATED BY ","
(CID,CNAME)
```

Save the above file in .ctl(control file) format.

Open cmd and type

```
SQLLDR HR/ADMIN CONTROL=C:\New\SUBJECT.CTL SKIP=1
```

Un-inserted rows can be view in .bad file which will be created automatically during compiling in the same path.

SQL Loader keywords

1) REPLACE

Before inserting data into table using sql loader the table must be empty. Instead of emptying table manually you could use ‘REPLACE’ keyword to delete all rows in a table before inserting in .ctl file.

```
LOAD DATA INFILE 'C:\New\COURSE.CSV'
REPLACE
INTO TABLE SQLL3
FIELDS TERMINATED BY ","
(CID,CNAME)
```

2) APPEND

This keyword is entirely opposite to ‘RELPACE’. This keyword is used to load data in a table even if the table already contains rows.

```
LOAD DATA INFILE 'C:\New\COURSE.CSV'
APPEND
INTO TABLE SQLL3
FIELDS TERMINATED BY ","
(CID,CNAME)
```

3) INFILE *

This signifies that the data is present at the end of the control file.

```
LOAD DATA INFILE *
APPEND
INTO TABLE SQLL3
FIELDS TERMINATED BY ","
(CID,CNAME)

BEGINDATA
001,ORACLE
002,UNIX
007,DOTNET
```

4) BEGINDATA

This indicates that this is the end of the control information and beginning of data.

```
LOAD DATA INFILE *
APPEND
INTO TABLE SQLL3
FIELDS TERMINATED BY ","
(CID,CNAME)

BEGINDATA
001,ORACLE
002,UNIX
007,DOTNET
```

5) FILLER POSITION

Sql loader allows filtering particular row and inserting same field in two different rows.

```
LOAD DATA
INFILE 'C:\New\COURSE2.CSV'
REPLACE

INTO TABLE EMPL1
FIELDS TERMINATED BY ","
(EMPNO,
EMPNAME,
DEPT_SKIP FILLER POSITION(1),
DISC,
DEPTNO,
DEPTNAME)
```

6) BOUND FILLER

It can be used if the skipped column's value will be required later again.

```
LOAD DATA
```

```
INFILE 'C:\New\COURSE3.CSV'
REPLACE

INTO TABLE EMPL1
FIELDS TERMINATED BY ","
(
REC_SKIP BOUNDFILLER,
TMP_SKIP BOUNDFILLER,
TML_SKIP BOUNDFILLER,
EMPNO"(:REC_SKIP||:TMP_SKIP||:TML_SKIP||:EMPNO)",
EMPNAME
)
```

7) WHEN

'WHEN' clause is used in SQL loader to impose condition during data loading.

```
LOAD DATA
INFILE 'C:\New\COURSE4.CSV'

REPLACE
INTO TABLE EMPL1
WHEN(01)='1'
FIELDS TERMINATED BY ","
(REC_SKIP FILLER POSITION(1),
EMPNO,
EMPNAME)
INTO TABLE DEPT1
WHEN(01)='2'
FIELDS TERMINATED BY ","
(REC_SKIP FILLER POSITION(1),
DEPTNO,
DEPTNAME)
```

8) SKIP

You can skip unwanted records using this command

```
OPTIONS (SKIP=1)
LOAD DATA INFILE 'C:\New\COURSE.CSV'
APPEND
INTO TABLE SQLL3
FIELDS TERMINATED BY ","
(CID,CNAME)
```

(0R)

```
SQLLDR HR/ADMIN CONTROL=C:\New\SUBJECT.CTL SKIP=1
```

9.) FIXED RECORD FORMAT

```
LOAD DATA
INFILE 'C:\New\COURSE5.CSV'
REPLACE
INTO TABLE EMPL1
(DEPTNAME POSITION(02:05) CHAR(4),
DISC POSITION(08:27) CHAR(20)
)
```

10.) VARAIBLE RECORD FORMAT

```
LOAD DATA
INFILE 'C:\New\COURSE5.CSV'
REPLACE
INTO TABLE EMPL1
(DEPTNAME POSITION(02:05) CHAR(4),
DISC POSITION(08:27) CHAR(5)
)
STREAM RECORD FORAMT
LOAD DATA
INFILE 'C:\New\COURSE7.CSV' "str '|\n'"
REPLACE
INTO TABLE EMPL1
```

```
FIELDS TERMINATED BY ';' TRAILING NULLCOLS
(DEPTNAME,
DISC
)

one line;hello dear world;|
two lines;Dear world,
hello!;|
```

Real Time Example

At some stage you need to insert bulk record into database which you have in .CSV file which cannot be done manually. This can be achieved by using SQL Loader to prevent time.

DBMS PROFILER

The built –in package allows you to turn on execution profiling in a session. Then why you run your code oracle uses table to keep track of the detailed information about how long each line in your code to execute. You can then run queries on these tables or much preferred use screens in products like TOAD or SQL Navigator to present the data in a clear graphical fashion.

dbms_profiler is oracle supplied package which use for tunning plsql application.
through this package we can tune our plsql procedure, trigger,funtions. and findout where plsql spent more time to execute.

DBMS_PROFILER Package:

Oracle 8i provides a new tool called PL/SQL Profiler. This is a powerful tool to analyze a Program unit execution and determine the runtime behavior. The results generated can then be evaluated to find out the hot areas in the code. This tool helps us identify performance bottlenecks, as well as where excess execution time is being spent in the code. The time spent in executing an SQL statement is also generated. This process is implemented with DBMS_PROFILER package.

The possible profiler statistics that are generated:

1. Total number of times each line was executed.
2. Time spent executing each line. This includes SQL statements.
3. Minimum and maximum duration spent on a specific line of code.
4. Code that is actually being executed for a given scenario.

DBMS_PROFILER.START_PROFILER

The DBMS_PROFILER.START_PROFILER tells Oracle to start the monitoring process. An identifier needs to be provided with each run that is used later to retrieve the statistics.

DBMS_PROFILER.STOP_PROFILER

The DBMS_PROFILER.STOP_PROFILER tells Oracle to stop the monitoring.

DBMS_PROFILER.FLUSH_DATA

The data collected for an execution is held in the memory. Calling the DBMS_PROFILER.FLUSH_DATA routine tells Oracle to save this data in profiler tables and clear the memory.

Example:

In Basic Oracle Installation "dbms_profiler" package is not created. We have to manually create this packages running below scripts.

conn with SYS user and run "PROFLOAD.SQL"
Located: $ORACLE_HOME/rdbms/admin folder

```
SQL> conn sys@hgc as sysdba
Enter password:
Connected.
SQL> @
C:\oraclexe\app\oracle\product\10.2.0\server\RDBMS\ADMIN\profload.sql

Package created.
Grant succeeded.
Synonym created.
Library created.
```

Package body created.

Testing for correct installation
SYS.DBMS_PROFILER successfully loaded.

PL/SQL procedure successfully completed.

Conn to application user and run "PROFTAB.SQL", this script create three tables.
Located: $ORACLE_HOME/rdbms/admin folder

1. PLSQL_PROFILER_RUNS
2. PLSQL_PROFILER_UNITS
3. PLSQL_PROFILER_DATA

SQL> conn scott/tiger@hgc
Connected.
SQL> @
C:\oraclexe\app\oracle\product\10.2.0\server\RDBMS\ADMIN\proftab.sql

Table created.
Comment created.
Table created.
Comment created.
Table created.
Comment created.
Sequence created.

How can use dbms_profiler ?
1. Connect with application user which you want to optimize.

Start dbms_profiler
SQL> exec dbms_profiler.start_profiler('Procedure_p1');

Information is store in memory so flash profiler to update their repository.
SQL> exec dbms_profiler.flush_data();

Stop dbms_profiler
SQL> exec dbms_profiler.stop_profiler();

View Information Generated by dbms_profiler
Query in below views
1. select * from plsql_profiler_runs;
2. select * from plsql_profiler_units
3. select * from plsql_profiler_data

DBMS UTILITY

Use this built-in function to calculate the elapsed time of your code down to the hundredth of a second. The scripts tmr-ot and plutmr.pkg (available on the book's web site) offer an interface to this function that allows you to use 'timers' (based on DBMS_UTILITY.GET_TIME) in your coed. They make it possible to time exactly how long a certain operation took to run and even to compare various implementations of the same requirements.

Example

```
DECLARE
N NUMBER;
BEGIN
N := DBMS_UTILITY.GET_TIME;
DBMS_OUTPUT.PUT_LINE(N);
END;
/
```

Output
835379

Example

```
DECLARE
TIME_BEFORE BINARY_INTEGER;
TIME_AFTER BINARY_INTEGER;
BEGIN
```

```
TIME_BEFORE := DBMS_UTILITY.GET_TIME;
P1(100);
TIME_AFTER := DBMS_UTILITY.GET_TIME;
DBMS_OUTPUT.PUT_LINE (TIME_AFTER - TIME_BEFORE);
END;
/
```

Output

0

PRAGMA

Pragma is a keyword in Oracle PL/SQL that is used to provide an instruction to the compile.

1) Pragama means force (like it says to compiler forcible do this operation)
2) It is used for committing the particular block.
3) We can't rollback this Pragama autonomous block

Types Of Pragmas :-

1 - AUTONOMOUS_TRANSACTION

2 - EXCEPTION_INIT

3 - RESTRICT_REFERENCES

4 - SERIALLY_REUSABLE

ORACLE FLASHBACK QUERY

Flashback query allows a user to view the data quickly and easily the way it was at a particular time in the past, even when it is modified and committed, be it a single row or the whole table.

Flashback technologies are applicable in repairing the following user errors.

1.) Erroneous or malicious DROP TABLE statements
2.) Erroneous or malicious update, delete or insert transactions
3.) Erroneous or malicious batch job or wide-spread application errors

Configuration before using Flashback Queries
In order to use this feature, the database instance has to be configured. Log on to the database where the test is to be performed and run the following command at the SQL prompts:

SQL> show parameter UNDO;

NAME TYPE VALUE

---------- ---------- ----------

undo_management string AUTO

undo_retention integer 900

undo_suppress_errors boolean TRUE

undo_tablespace string UNDO_TBSPC

This command displays all the necessary parameters for using the Flashback Queries. The value for UNDO_RETENTION is set to 900 seconds in this example which is the default value and represents at least how long the system retains undo. UNDO_RETENTION and UNDO_TABLESPACE are dynamic parameters, but UNDO_MANAGEMENT is not, requiring shut down and re-start of database instance in order for automatic undo management to take effect.

In addition to the above your DBA will have to grant:

FLASHBACK privilege to the user for all or a subset of objects
Execute privileges on the dbms_flashback package

Using the Flashback Query with AS OF clause:

Suppose we want to recover data we have accidentally deleted for some of the employees from the EMPLOYEE table and have committed the transaction.

```
SQL> INSERT INTO EMPLOYEE_TEMP
   (SELECT * FROM EMPLOYEE AS OF TIMESTAMP ('13-SEP-04 8:50:58','DD-MON-YY HH24: MI: SS'
```

Using a point in time is way of going back, another way of telling the system how far to go back is the use of SCN - System Change Number. The procedure is the same as earlier, trying to recover lost data using the DBMS_FLASHBACK utility. Only this time instead of using the time we are using the system change number - SCN to enter the flashback mode. This SCN number can be obtained before the transaction is initiated by using the GET_SYSTEM_CHANGE_NUMBER function of the DBMS_FLASHBACK utility as follows.

```
SQL> select DBMS_FLASHBACK. GET_SYSTEM_CHANGE_NUMBER from dual;
```

```
SQL> INSERT INTO EMPLOYEE_TEMP
        (SELECT * FROM EMPLOYEE AS OF SCN 10280403339);
```

Oracle 10G has enhanced the flashback feature further and has turned it into a much more powerful feature by introducing numerous additions. Some of the more common ones are discussed here.

1.) Flashback Table
2.) Flashback Drop
3.) Flashback Database
4.) Flashback Versions Query
5.) Flashback Transaction Query

1.) Flashback Table

Just like the flashback query helps retrieve rows of a table, FLASHBACK TABLE helps restore the state of a table to a certain point in time even if a table structure changed has occurred since then. The following simple command will take us to the table state at the specified timestamp.

```
SQL> FLASHBACK TABLE Employee TO
        TIMESTAMP ('13-SEP-04 8:50:58','DD-MON-YY HH24: MI: SS');
```

Not only does this command restore the tables but also the associated objects like indexes, constraints etc.

2.) Flashback Drop

So far we have recovered the lost data to a particular point-in-time back into a table that exists in the database. Oracle 10g has provided another useful feature term as the Flashback drop. For our example if a DROP TABLE has been issued for the table EMPLOYEE we can still restore the whole table by issuing the following command.

```
SQL> FLASHBACK TABLE EMPLOYEE TO BEFORE DROP;
```

Bringing back dropped tables could not be any easier than this.

3.) RECYCLE BIN

It is worthwhile to take a little detour and familiarize ourselves with the feature in Oracle 10g that enabls us to flashback. Oracle has introduced the RECYCLE BIN which is a logical entity to hold all the deleted objects and works exaclty like the recylce bin provided in Windows operating system for example. All the deleted objects are kept n the recylce bin, these objects can be retrieved from the recycle bin or deleted permanently by using the PURGE command. Either an indivisual object like a table or an index can be deleted from the recycle bin:

```
SQL> PURGE TABLE Employee;
```

or the whole recylce bin can be 'emptied out' by using the PURGE command:

SQL> PURGE recyclebin;

If you take a look at the contents of the recycle bin using the following query,

SQL> select OBJECT_NAME, ORIGINAL_NAME, TYPE from user_recyclebin;

OBJECT_NAME ORIGINAL_NAME TYPE

BIN$G/gHMigrTRqHQukZSIpSLw==$0 EMPOLYEE TABLE

BIN$1UiHeUR7SymGHo20pTfGXA==$0 EMPLOYEE TABLE

BIN$6d6677f5T+K++npt+5p/jQ==$0 EMP_IDX1 INDEX

4.) Flashback database

Flash Recovery Area created by the DBA, is the allocation of space on the disk to hold all the recovery related files in one, centralized place. Flash Recovery Area contains the Flashback Logs, Redo Archive logs, backups files by RMAN and copies of control files. The destination and the size of the recovery area are setup using the db_recovery_file_dest and b_recovery_file_dest_size initializatin parameters. Now when the setup is complete, let's see how the flashback database is used.

For this test suppose that a transaction ran that made significant changes ti the database, yet this is not what the user intended. Going back and retrieving individual objects and then recovernig and restoring the original data can be a very extensive, yet timeconsuming and error-prone exercise. It is time to use the FLASHBACK DATABASE.

First the flashback is enabled to make Oracle database enter the flashback mode. The database must be mounted Exclusive and not open. The database

has to be in the ARCHIVELOG MODE before we can use this feature. This is shown as below.

SQL> ALTER DATABASE ARCHIVELOG;
Now startup the database in EXCLUSIVE mode.

SQL> SHUTDOWN IMMEDIATE;
SQL> STARTUP MOUNT EXCLUSIVE
Now enter the flashback mode (the database should not be open at this time)

> ALTER DATABASE FLASHBACK ON;
Issue the flashback command and take the database to the state it was in, one hour ago.

SQL> Flashback database to timestamp sysdate-(1/24);
After the system comes back with FLASHBACK COMPLETE, open the database.

SQL> ALTER DATABASE OPEN RESETLOGS;

Now if you select from any of the tables that were affected, you will see that the affected tables are in the original state, i.e. an hour ago. And once again, we have the option of using SCN instead of timestamp.

PL/SQL_WARNINGS
Compile-Time Warnings:

Oracle can now produce compile-time warnings when code is ambiguous or inefficient be setting the PLSQL_WARNINGS parameter at either instance or session level. The categories ALL, SEVERE, INFORMATIONAL and PERFORMANCE can be used to alter the type of warnings that are produced.

```
-- Instance and session level.
ALTER SYSTEM SET PLSQL_WARNINGS='ENABLE:ALL';
ALTER SESSION SET PLSQL_WARNINGS='DISABLE:PERFORMANCE';

-- Recompile with extra checking.
```

```
ALTER PROCEDURE hello COMPILE
PLSQL_WARNINGS='ENABLE:PERFORMANCE';

-- Set mutiple values.
ALTER SESSION SET
PLSQL_WARNINGS='ENABLE:SEVERE','DISABLE:PERFORMANCE','DISABLE:INFO
RMATIONAL';

-- Use the DBMS_WARNING package instead.
EXEC DBMS_WARNING.SET_WARNING_SETTING_STRING('ENABLE:ALL'
,'SESSION');
```

The current settings associated with each object can be displayed using the [USER|DBA|ALL]_PLSQL_OBJECT_SETTINGS views.

To see a typical example of the warning output run the following code.

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';

CREATE OR REPLACE PROCEDURE test_warnings AS
  l_dummy  VARCHAR2(10) := '1';
BEGIN
  IF 1=1 THEN
    SELECT '2'
    INTO   l_dummy
    FROM   dual;
  ELSE
    RAISE_APPLICATION_ERROR(-20000, 'l_dummy != 1!');
  END IF;
END;
/

SP2-0804: Procedure created with compilation warnings

SHOW ERRORS

LINE/COL ERROR
```

-------- --------------------------
9/5 PLW-06002: Unreachable code

Warning Category:

1.) -- severe
SELECT dbms_warning.get_category(5000)
FROM dual;

2.) -- informational
SELECT dbms_warning.get_category(6002)
FROM dual;

3.) -- performance
SELECT dbms_warning.get_category(7203)
FROM dual;