CREATION OF A DATABASE

Creation, altering and dropping of tables and inserting rows into a table (Use constraints while creating tables) examples using SELECT command.

Following tables (Relations) are considered for the lab purpose.

- SAILORS (SID:INTEGER, SNAME:STRING, RATING:INTEGER, AGE:REAL)
- BOATS (BID:INTEGER, BNAME:STRING, COLOR:STRING)
- RESERVES (SID:INTEGER, BID:INTEGER, DAY:DATE)

Creating Tables :-

The CREATE TABLE command is used to create the table (relation) in SQL.

CREATE TABLE TABLENAME (ATT_NAME1 DATATYPE, ATT_NAME2 DATATYPE, ATT_NAME3 DATATYPE, …..);

SQL> CREATE TABLE SAILORS (SID NUMBER (5), SNAME VARCHAR2(30), RATING NUMBER(5), AGE NUMBER(4,2));

Data Types :- Oracle supports following types of data types.

- CHAR (SIZE) :- Fixed length character data of length SIZE bytes. The maximum length is 255 bytes in Oracle 7 and 2000 bytes in Oracle 8 onwards. Default and minimum size is 1 byte.
- VARCHAR2(SIZE) :- Variable length character string having maximum length SIZE bytes. The maximum length 2000 bytes in Oracle 7 and 4000 bytes in Oracle 8 onwards. The minimum size is 1
- NUMBER(L) :- Numeric data with number of digits L.
- NUMBER(L, D) :- Numeric data with total number of digits L and number of digits D after decimal point.
- DATE :- Valid date range. The date ranges from January 1, 4712 BC to December 31, 9999 AD.
- LONG :- Character data of variable length which stores upto 2 Gigabytes of data. (A bigger version the VARCHAR2 datatype).
- INTEGER :- Integer type of Data. It is actually a synonym for NUMBER(38)

- FLOAT :- Floating point type of Data. Very similar to NUMBER it stores zero, positive, and negative floating-point numbers.

Along with these types of data types, Oracle supports other types of data types like TIMESTAMP, RAW, ROWID, CLOB, NCLOB, BLOB, XMLType, etc.

Primary Key & Foreign Key :- Consider the Sailors relation and the constraint that no two sailors have the same SID. This type of constraint can be defined using **Primary Key**, which gives uniqueness for the value of attribute defined (Eg. SID).

Similarly, a sailor can't reserve a boat unless he/she is a valid sailor i.e. the SID of Reserves relation must available in the Sailors relation. This type of constraint can be defined using **Foreign Key**, which gives the existence of the value of attribute in one relation is depends on value in another relation.

We can use Primary Key or/and Foreign Key constraint while creating table.

Creating tables with Primary Key

CREATE TABLE TABLENAME (ATT_NAME1 DATATYPE, ATT_NAME2 DATATYPE, ATT_NAME3 DATATYPE ….., **PRIMARY KEY(ATT_NAMES)**);

SQL> CREATE TABLE SAILORS (SID NUMBER(5), SNAME VARCHAR2(30), RATING NUMBER(5), AGE NUMBER(4,2), , **PRIMARY KEY(SID)**);

ALTERNATE METHOD

CREATE TABLE TABLENAME (ATT_NAME1 DATATYPE **PRIMARY KEY**, ATT_NAME2 DATATYPE, ATT_NAME3 DATATYPE …..);

Creating tables with Foreign Key

CREATE TABLE TABLENAME (ATT_NAME1 DATATYPE, ATT_NAME2 DATATYPE, ATT_NAME3 DATATYPE ….., **FOREIGN KEY (ATT_NAME) REFERENCES TABLENAME2)**);

ALTERNATE METHOD

CREATE TABLE TABLENAME (ATT_NAME1 DATATYPE, ATT_NAME2 DATATYPE **REFERENCES TABLENAME2(ATT_NAME),**ATT_NAME3 DATATYPE);

SQL> CREATE TABLE RESERVES (SID NUMBER(5), BID NUMBER(5), DAY DATE,

FOREIGN KEY (SID) REFERENCES (SAILORS));

The following example gives the complete definition to create Reserves table (Defines Primary Key as well as Foreign Keys).

SQL> CREATE TABLE RESERVES (SID NUMBER(5), BID NUMBER(5), DAY DATE, PRIMARY KEY (SID, BID, DAY), FOREIGN KEY (SID) REFERENCES (SAILORS) , FOREIGN KEY (BID) REFERENCES (BOATS));

Similar way we can create Sailors as well as Boats table using Primary Key constraint.

Creating table with some Constraint :- Suppose we want to add rule for rating of the sailors - "Rating should be between 1 to 10" while creating table then we can use following command.

SQL> CREATE TABLE SAILORS (SID NUMBER(5), SNAME VARCHAR2(30), RATING NUMBER(5), AGE NUMBER(4,2), , PRIMARY KEY(SID), **CHECK (RATING >=1 AND RATING <=10)**);

Deleting Table :- The table along with its definition & data can be deleted using following command.

DROP TABLE <TABLENAME>;

SQL> DROP TABLE SAILORS;

Adding & Deleting the Attributes and Constraints to the Table :- To add the attribute to a existing relation we can use ALTER TABLE Command. ALTER TABLE <TABLENAME> ADD COLUMN ATT_NAME DATATYPE;

SQL> ALTER TABLE SAILORS ADD COLUMN SALARY NUMBER(7,2);

To remove the attribute from an existing relation we can use following Command.

ALTER TABLE <TABLENAME> DROP COLUMN ATT_NAME;

SQL> ALTER TABLE SAILORS DROP COLUMN SALARY;

To add the constraint to existing relation we can use ALTER TABLE Command.

ALTER TABLE <TABLENAME> ADD CONSTRAINT <CON_NAME> <CON_DEFINITION>;

SQL> ALTER TABLE SAILORS ADD CONSTRAINT RATE CHECK (RATING >= 1 AND RATING <=10);

Similarly we can add primary key or foreign key constraint.

To delete the constraint to existing relation we can use following Command.

DROP CONSTRAINT <CON_NAME>;

SQL> DROP CONSTRAINT RATE;

Similarly we can drop primary key or foreign key constraint.

Adding data to the Table :- We can add data to table by using INSERT INTO command. While adding the data to the table we must remember the order of attributes as well as their data types as defined while creating table. The syntax is as follows.

INSERT INTO <TABLENAME> VALUES (VALUE1, VALUE2, VALUE3, …..);

SQL> INSERT INTO SAILORS VALUES (1, 'Rajesh', 10, 30);

But sometimes while adding data we may not remember the exact order or sometimes we want to insert few values then we can use following format to add data to a table.

INSERT INTO <TABLENAME> (ATT_NAME1, ATT_NAME2, ATT_NAME3, …..) VALUES (VALUE1, VALUE2, VALUE3, …..);

SQL> INSERT INTO SAILORS (SNAME, SID, AGE, RATING) VALUES ('Rajesh', 1, 30, 10);

If we want the data to be entered from the keyboard,we can use the following format

INSERT INTO <TABLENAME> VALUES (&ATT_NAME1, & ATT_NAME2, & ATT_NAME3, …..);

By using any one of these methods we can add records or data to Sailors, Boats as well as Reserves Table.

To see the records :- To view all records present in the table.

SELECT * FROM <TABLENAME>

SQL> SELECT * FROM SAILORS;

To delete the record(s) :- To delete all records from table or a single/multiple records which matches the given condition, we can use DELETE FROM command as follows.

DELETE FROM <TABLENAME> WHERE <CONDITION>;

SQL> DELETE FROM SAILORS WHERE SNAME = 'Rajesh';

To delete all records from the table

DELETE FROM <TABLENAME>;

SQL> DELETE FROM SAILORS;

To change particular value :- We can modify the column values in an existing row using the UPDATE command.

UPDATE <TABLENAME> SET ATT_NAME = NEW_VALUE WHERE CONDITION;

SQL> UPDATE SAILORS SET RATING = 9 WHERE SID = 1;

To update all records without any condition.

SQL> UPDATE SAILORS SET RATING = RATING + 1;

Simple Queries on the Tables :- The basic form of an SQL query is: SELECT <SELECT_LIST>FROM <TABLE_LIST> WHERE <CONDITION>; Q1)

Display names & ages of all sailors.

SQL> SELECT SNMAE, AGE FROM SAILORS;

Queries on multiple tables

Q2) Find the names of sailors who have reserved boat number 123.

SQL> SELECT SNAME FROM SAILORS S, RESERVES R WHERE S.SID = R.SID AND R.BID = 123;

Write queries for

1) Find SIDs of sailors who have reserved Pink Boat;
2) Find the color of the boats reserved by Rajesh.
3) Find names of the sailors who have reserved at least one boat.

Queries (along with sub Queries) using ANY, ALL, IN,
EXISTS, NOTEXISTS, UNION, INTERSET, Constraints.

DISTINCT Keyword :- The DISTINCT keyword eliminates the duplicate tuples from the result records set.

Ex:- Find the Names and Ages of all sailors.

SQL> SELECT DISTINCT S.SNAME, S.AGE FROM SAILORS S;

The answer is a set of rows, each of which is a pair (sname, age). If two or more sailors have the same name and age, the answer still contains just one pair with that name and age.

UNION, INTERSECT, EXCEPT (MINUS) :- SQL provides three set-manipulation constructs that extend the basic query form. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference.

SQL supports these operations under the names UNION, INTERSECT and MINUS.

Note that UNION, INTERSECT, and MINUS can be used on any two tables that are union-compatible, that is, have the same number of columns and the columns, taken in order, have the same types.

UNION :- It is a set operator used as alternative to **OR** query.

Here is an example of Query using **OR.**

Ex:- Find the names of sailors who have reserved a red or a green boat.

SQL> SELECT S.SNAME FROM SAILORS S, RESERVES R, BOATS B WHERE S.SID = R.SID AND R.BID = B.BID AND (B.COLOR = 'RED' **OR** B.COLOR = 'GREEN');

Same query can be written using **UNION** as follows.

SQL> SELECT S.SNAME FROM SAILORS S, RESERVES R, BOATS B WHERE S.SID = R.SID AND R.BID = B.BID AND B.COLOR = 'RED' **UNION** SELECT S2.SNAME FROM SAILORS S2, BOATS B2, RESERVES R2 WHERE S2.SID = R2.SID AND R2.BID = B2.BID AND B2.COLOR = 'GREEN';

This query says that we want the union of the set of sailors who have reserved red boats and the set of sailors who have reserved green boats.

INTERSECT :- It is a set operator used as alternative to **AND** query. Here is an example of Query using **AND.**

Ex:- Find the names of sailor's who have reserved both a red and a green boat.

SQL> SELECT S.SNAME FROM SAILORS S, RESERVES R1, BOATS B1, RESERVES R2, BOATS B2 WHERE S.SID = R1.SID AND R1.BID = B1.BID AND S.SID = R2.SID AND R2.BID = B2.BID AND B1.COLOR='RED' AND B2.COLOR = 'GREEN';

Same query can be written using **INTERSECT** as follows.

SQL> SELECT S.SNAME FROM SAILORS S, RESERVES R, BOATS B WHERE S.SID = R.SID AND R.BID = B.BID AND B.COLOR = 'RED' INTERSECT **SELECT** S2.SNAME FROM SAILORS S2, BOATS B2, RESERVES R2 WHERE S2.SID = R2.SID AND R2.BID = B2.BID AND B2.COLOR = 'GREEN';

EXCEPT (MINUS) :- It is a set operator used as set-difference. Our next query illustrates the set-difference operation.

Ex:- Find the sids of all sailor's who have reserved red boats but not green boats.

SQL> SELECT S.SID FROM SAILORS S, RESERVES R, BOATS B WHERE S.SID = R.SID AND R.BID = B.BID AND B.COLOR = 'RED' **MINUS** SELECT S2.SID FROM SAILORS S2, RESERVES R2, BOATS B2 WHERE S2.SID = R2.SID AND R2.BID = B2.BID AND B2.COLOR = 'GREEN';

Same query can be written as follows. Since the Reserves relation contains sid information, there is no need to look at the Sailors relation, and we can use the following simpler query

SQL> SELECT R.SID FROM BOATS B, RESERVES R WHERE R.BID = B.BID AND B.COLOR = 'RED' **MINUS** SELECT R2.SID FROM BOATS B2, RESERVES R2 WHERE R2.BID = B2.BID AND B2.COLOR = 'GREEN';

NESTED QUERIES:- For retrieving data from the tables we have seen the simple & basic queries. These queries extract the data from one or more tables. Here we are going to see some complex & powerful queries that enables us to retrieve the data in desired manner. One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a subquery.

IN Operator :- The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested.

Ex:- Find the names of sailors who have reserved boat 103.

SQL> SELECT S.SNAME FROM SAILORS S WHERE S.SID **IN** (SELECT R.SID FROM RESERVES R WHERE R.BID = 103);

NOT IN Operator :- The NOT IN is used in a opposite manner to IN.

Ex:- Find the names of sailors who have not reserved boat 103.

SQL> SELECT S.SNAME FROM SAILORS S WHERE S.SID **NOT IN** (SELECT R.SID FROM RESERVES R WHERE R.BID = 103);

EXISTS Operator :- This is a Correlated Nested Queries operator. The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty, an implicit comparison with the empty set.

Ex:- Find the names of sailors who have reserved boat number 103.

SQL> SELECT S.SNAME FROM SAILORS S WHERE **EXISTS** (SELECT * FROM RESERVES R WHERE R.BID = 103 AND R.SID = S.SID);

NOT EXISTS Operator :- The NOT EXISTS is used in a opposite manner to EXISTS.

Ex:- Find the names of sailors who have not reserved boat number 103.

SQL> SELECT S.SNAME FROM SAILORS S WHERE **NOT EXISTS** (SELECT * FROM RESERVES R WHERE R.BID = 103 AND R.SID = S.SID);

Set-Comparison Operators:- We have already seen the set-comparison operators EXISTS, IN along with their negated versions. SQL also supports **op ANY** and **op ALL**, where **op** is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}. Following are the example which illustrates the use of these Set-Comparison Operators.

op ANY Operator :- It is a comparison operator. It is used to compare a value with any of element in a given set.

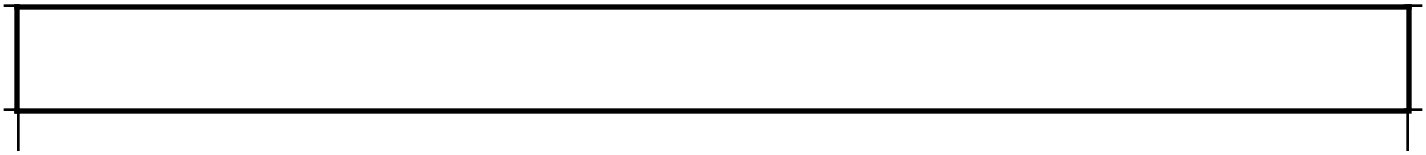
Ex:- Find sailors whose rating is better than some sailor called Rajesh.

SQL> SELECT S.SID FROM SAILORS S WHERE S.RATING **> ANY** (SELECT S2.RATING FROM SAILORS S2 WHERE S2.SNAME = ' RAJESH ');

op ALL Operator :- It is a comparison operator. It is used to compare a value with all the elements in a given set.

Ex:- Find the sailor's with the highest rating using ALL.

SQL> SELECT S.SID FROM SAILORS S WHERE S.RATING >= **ALL** (SELECT S2.RATING FROM SAILORS S2)



Queries using Aggregate functions (COUNT, SUM, AVG, MAX and MIN), GROUP BY, HAVING and Creation and dropping of Views.

AGGREGATE Functions :- In addition to simply retrieving data, we often want to perform some computation or summarization. We now consider a powerful class of constructs for computing aggregate values such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT (A) :- The number of values in the A column.

Or COUNT (DISTINCT A): The number of unique values in the A column.

Ex:- 1) To count number SIDs of sailors in Sailors table

SQL> SELECT **COUNT** (SID) FROM SAILORS;

2) To count numbers of boats booked in Reserves table.

SQL> SELECT **COUNT** (DISTINCT BID) FROM RESERVES;

3) To count number of Boats in Boats table.

SQL> SELECT **COUNT** (*) FROM BOATS;

2. SUM (A) :- The sum of all values in the A column.

Or SUM (DISTINCT A): The sum of all unique values in the A column.

Ex:- 1) To find sum of rating from Sailors

SQL> SELECT **SUM** (RATING) FROM SAILORS;

2) To find sum of distinct age of Sailors (Duplicate ages are eliminated).

SQL> SELECT **SUM** (DISTINCT AGE) FROM SAILORS;

3. AVG (A) :- The average of all values in the A column.

Or AVG (DISTINCT A): The average of all unique values in the A column.

Ex:- 1) To display average age of Sailors.

SQL> SELECT **AVG** (AGE) FROM SAILORS;

2) To find average of distinct age of Sailors (Duplicate ages are eliminated).

SQL> SELECT **AVG** (DISTINCT AGE) FROM SAILORS;

4. MAX (A) :- The maximum value in the A column.

Ex:- To find age of Oldest Sailor.

SQL> SELECT **MAX** (AGE) FROM SAILORS;

5. MIN (A) :- The minimum value in the A column.

Ex:- To find age of Youngest Sailor.

SQL> SELECT **MIN** (AGE) FROM SAILORS;

Note that it does not make sense to specify DISTINCT in conjunction with MIN or MAX (although SQL does not preclude this).

Write the following queries using Aggregate Functions.

1) Find the average age of sailors with a rating of 10.

2) Count the number of different sailor names.

3) Find the name and age of the oldest sailor.

4) Count the number of Sailors.

5) Find the names of sailors who are older than the oldest sailor with a rating of 10.

ORDER BY Clause :- The ORDER BY keyword is used to sort the result-set by a specified column. The ORDER BY keyword sorts the records in ascending order by default (we can even use ASC keyword). If we want to sort the records in a descending order, we can use the DESC keyword. The general syntax is

SELECT ATT_LIST FROM TABLE_LIST ORDER BY ATT_NAMES [ASC | DESC];

Ex:- 1) Display all the sailors according to their ages.

SQL> SELECT * FROM SAILORS ORDER BY AGE;

2) Display all the sailors according to their ratings (topper first).

SQL> SELECT * FROM SAILORS ORDER BY RATING DESC;

3) Displays all the sailors according to rating, if rating is same then sort according to age.

SQL> SELECT * FROM SAILORS ORDER BY RATING, AGE;

Write the query

1) To display names of sailors according to alphabetical order.

2) Displays all the sailors according to rating (Topper First), if rating is same then sort according to age (Older First).
3) Displays all the sailors according to rating (Topper First), if rating is same then sort according to age (Younger First).
4) Displays all the sailors according to rating (Lower Rating First), if rating is same then sort according to age (Younger First).

GROUP BY and HAVING Clauses :- Thus far, we have applied aggregate operations to all (qualifying) rows in a relation. Often we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance. For this purpose we can use Group by clause.

GROUP BY:- Group by is used to make each a number of groups of rows in a relation, where the number of groups depends on the relation instances. The general syntax is
SELECT [DISTINCT] ATT_LIST FROM TABLE_LIST WHERE CONDITION **GROUP BY** GROUPING_LIST;
Ex:- Find the age of the youngest sailor for each rating level.
SQL> SELECT S.RATING, MIN (S.AGE) FROM SAILORS S **GROUP BY** S.RATING;

HAVING :- The extension of GROUP BY is HAVING clause which can be used to specify the qualification over group. The general syntax is
SELECT [DISTINCT] ATT_LIST FROM TABLE_LIST WHERE CONDITION GROUP BY GROUPING_LIST **HAVING** GROUP_CONDITIION;
Ex :- Find the age of youngest sailor with age >= 18 for each rating with at least 2 such sailors.
SQL> SELECT S.RATING, MIN (S.AGE) AS MINAGE FROM SAILORS S WHERE S.AGE >= 18 GROUP BY S.RATING **HAVING** COUNT (*) > 1;

Write following queries in SQL.
1) For each red boat; find the number of reservations for this boat.
2) Find the average age of sailors for each rating level that has at least two sailors.
3) Find those ratings for which the average age of sailors is the minimum over all ratings.

VIEWS :- A view is a table whose rows are not explicitly stored in the database but are computed as needed from a view definition. The views are created using **CREATE VIEW** command.

Ex :- Create a view for Expert Sailors (A sailor is a Expert Sailor if his rating is more than 7).

SQL> CREATE VIEW EXPERTSAILOR AS SELECT SID, SNAME, RATING FROM SAILORS WHERE RATING > 7;

Now on this view we can use normal SQL statements as we are using on Base tables.

Eg:- Find average age of Expert sailors.

SQL> SELECT AVG (AGE) FROM EXPERTSAILOR;

Write the following queries on Expert Sailor View.

1) Find the Sailors with age > 25 and rating equal to 10.
2) Find the total number of Sailors in Expert Sailor view.
3) Find the number of Sailors at each rating level (8, 9, 10).
4) Find the sum of rating of Sailors.
5) Find the age of Oldest as well as Youngest Expert Sailor.

If we decide that we no longer need a view and want to destroy it (i.e. removing the definition of view) we can drop the view. A view can be dropped using the **DROP VIEW** command.

To drop the ExpertSailor view.

SQL> DROP VIEW EXPERTSAILOR;

Date Functions :-

1) SYSDATE :- Displays the system date for a system.

Select sysdate from dual;

PL/SQL PROGRAMMING

Procedural Language/Structured Query Language (PL/SQL) is an extension of SQL.

Basic Syntax of PL/SQL

```
DECLARE
 /* Variables can be declared here */
BEGIN
   /* Executable statements can be written here */
EXCEPTION
/* Error handlers can be written here. */
END;
```

Steps to Write & Execute PL/SQL

¾ As we want output of PL/SQL Program on screen, before Starting writing anything type (Only Once per session)

```
SQL> SET SERVEROUTPUT ON
```

¾ To write program, use Notepad through Oracle using ED command.

```
SQL> ED ProName
```

Type the program Save & Exit.

¾ To Run the program

```
SQL> @ProName
```

Ex :- PL/SQL to find addition of two numbers

```
DECLARE
A INTEGER := &A;
B INTEGER := &B;
C INTEGER;
BEGIN
C :=  A + B;
```

```
DBMS_OUTPUT.PUT_LINE('THE SUM IS '||C);
END;
/
```

Decision making with IF statement :- The general syntax for the using IF--ELSE statement is

```
IF(TEST_CONDITION) THEN
  SET OF STATEMENTS
ELSE
  SET OF STATEMENTS
END IF;
```

For Nested IF—ELSE Statement we can use IF--ELSIF—ELSE as follows

```
IF(TEST_CONDITION) THEN
  SET OF STATEMENTS
ELSIF (CONDITION)
  SET OF STATEMENTS
END IF;
```

Ex:- Largest of three numbers.

This program can be written in number of ways, here are the two different ways to write the program.

1)

```
DECLARE
 A NUMBER := &A;
B NUMBER := &B;
C NUMBER := &C;
BIG NUMBER;
BEGIN
 IF (A > B) THEN
```

```
  BIG := A;
 ELSE
  BIG := B;
 END IF;
 IF(BIG < C ) THEN
  DBMS_OUTPUT.PUT_LINE('BIGGEST OF A, B AND C IS ' || C);
 ELSE
    DBMS_OUTPUT.PUT_LINE('BIGGEST OF A, B AND C IS ' || BIG);
 END IF;
END;
/
```

2)

```
DECLARE
 A NUMBER := &A;
B  NUMBER  := &B;
C  NUMBER  := &C;
BEGIN
 IF (A > B AND A > C) THEN
 DBMS_OUTPUT.PUT_LINE('BIGGEST IS ' || A);
 ELSIF (B > C) THEN
 DBMS_OUTPUT.PUT_LINE('BIGGEST IS ' || B);
 ELSE
  DBMS_OUTPUT.PUT_LINE('BIGGEST IS ' || C);
 END IF;
END;
/
```

LOOPING STATEMENTS:- For executing the set of statements repeatedly we can use loops. The oracle supports number of looping statements like GOTO, FOR, WHILE & LOOP. Here is the syntax of these all the types of looping statements.

- **GOTO STATEMENTS**

```
<<LABEL>>
SET OF STATEMENTS
GOTO LABEL;
```

- **FOR LOOP**

```
FOR <VAR> IN [REVERSE] <INI_VALUE>..<END_VALUE>
    SET OF STATEMENTS
END LOOP;
```

- **WHILE LOOP**

```
WHILE (CONDITION) LOOP
    SET OF STATEMENTS
END LOOP;
```

- **LOOP STATEMENT**

```
LOOP
    SET OF STATEMENTS
IF (CONDITION) THEN
    EXIT
    SET OF STATEMENTS
END LOOP;
```

While using LOOP statement, we have take care of EXIT condition, otherwise it may go into infinite loop.

Example :- Here are the example for all these types of looping statement where each program prints numbers 1 to 10.

GOTO EXAMPLE

```
DECLARE
```

```
 I INTEGER := 1;
BEGIN
 <<OUTPUT>>
DBMS_OUTPUT.PUT_LINE(I);
I := I + 1;
IF I<=10 THEN
GOTO OUTPUT;
END IF;
END;
/
```

FOR LOOP EXAMPLE

```
BEGIN
 FOR I IN 1..10 LOOP
DBMS_OUTPUT.PUT_LINE(I);
END LOOP;
END;
/
```

WHILE EXAMPLE

```
DECLARE
 I INTEGER := 1;
BEGIN
 WHILE(I<=10) LOOP
DBMS_OUTPUT.PUT_LINE(I);
I := I + 1;
END LOOP;
END;
/
```

LOOP EXAMPLE

```
DECLARE
 I INTEGER := 1;
BEGIN
 LOOP
DBMS_OUTPUT.PUT_LINE(I);
I := I + 1;
EXIT WHEN I=11;
END LOOP;
END;
/
```

DATA TYPES

Already we know following data types.

NUMBER, INTEGER, VARCHAR2, DATE, BOOLEAN, etc. Now let's see few more data types that are useful for writing PL/SQL programs in Oracle.

- **%TYPE :-** %TYPE is used to give data type of predefined variable or database column.

  Eg:- itemcode Number(10);

  icode itemcode%Type;

  The database column can be used as

  id Sailors.sid%type
- **%ROWTYPE :-** %rowtype is used to provide record datatype to a variable. The variable can store row of the table or row fetched from the cursor.
- **Eg:-** If we want to store a row of table Sailors then we can declare variable as

  Sailors %Rowtype

Comments :- In Oracle we can have two types of comments i.e Single Line & Multiline comments.

- Single line comment :- It starts with --.

  -- Comment here

- Multiline comment is same as C/C++/JAVA comments where comments are present in the pair of /* & */.

  /* Comment here */

Inserting values to table :- Here is the example for inserting the values into a database through PL/SQL Program. Remember that we have to follow all the rules of SQL like Primary Key Constraints, Foreign Key Constraints, Check Constraints, etc.

Ex:- Insert the record into Sailors table by reading the values from the Keyboard.

```
DECLARE
SID NUMBER (5):=&SID;
SNAME VARCHAR2(30):='&SNAME';
RATING NUMBER(5):=&RATING;
AGE NUMBER(4,2):=&AGE;
BEGIN
INSERT INTO SAILORS VALUES(SID, SNAME, RATING, AGE);
END;
/
```

Reading from table

```
DECLARE
SID VARCHAR2(10); -- or can be defined SID Sailors.SID%Type
SNAME VARCHAR2(30);
RATING NUMBER(5);
AGE NUMBER(4,2);
BEGIN
SELECT SID, SNAME, RATING, AGE INTO SID, SNAME, RATING, AGE FROM SAILORS WHERE SID='&SID';
DBMS_OUTPUT.PUT_LINE(SID || ' '|| SNAME || ' '|| RATING ||' '|| AGE );
END;
/
```

Some Points regarding SELECT --- INTO

We have to ensure that the SELECT….INTO statement should return one & only one row. If no row is selected then exception NO_DATA_FOUND is raised. If more than one row is selected then exception TOO_MANY_ROWS is raised.

To handle the situation where no rows selected or so many rows selected we can use Exceptions. We have two types of exception, User-Defined and Pre-Defined Exceptions.

Program with User-Defined Exception

```
DECLARE
N INTEGER:=&N;
A EXCEPTION;
B EXCEPTION;
BEGIN
IF MOD(N,2)=0 THEN
 RAISE A;
ELSE RAISE
B; END IF;
EXCEPTION
WHEN A THEN
DBMS_OUTPUT.PUT_LINE('THE INPUT IS EVEN.....');
WHEN B THEN
DBMS_OUTPUT.PUT_LINE('THE INPUT IS ODD.....');
END;
/
```

Program with Pre-Defined Exception

```
DECLARE
SID VARCHAR2(10);
```

```
BEGIN
SELECT SID INTO SID FROM SAILORS WHERE SNAME='&SNAME';
DBMS_OUTPUT.PUT_LINE(SID);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('No Sailors with given SID found');
WHEN TOO_MANY_ROWS THEN
DBMS_OUTPUT.PUT_LINE('More than one Sailors with same name found');
END;
/
```

Cursors

Oracle uses temporary work area cursor for storing output of an SQL statement. Cursors are defined as

```
CURSOR C1 IS SELECT SID, SNAME, RATING, AGE FROM SAILORS;
```

OR

```
CURSOR C1 IS SELECT * FROM SAILORS;
```

Generally while using cursors we have to Open the cursor then extract one row (record) from the cursor using Fetch operation, do the necessary operations on the Record. After completing the work Close the cursor. But if we want to do automatic opening & closing to cursor then we can use FOR loop in cursor.

FOR loop in Cursor

The cursor FOR loop gives easy way to handle the Cursor. The FOR loop opens the Cursor, fetches rows and closes the cursor after all rows are processed.

For **Eg:**

```
FOR Z IN C1 LOOP
- - - - - - -
END LOOP;
```

The cursor FOR loop declares Z as record, which can hold row, returned from cursor.

Example using Cursor

```
DECLARE
 CURSOR C1 IS SELECT * FROM SAILORS;
BEGIN
FOR Z IN C1 LOOP
DBMS_OUTPUT.PUT_LINE
(Z.SID || ' ' || Z.SNAME);
END LOOP;
END;
/
```

Same example using While

```
DECLARE
 CURSOR C1 IS SELECT * FROM SAILORS;
Z C1%ROWTYPE;
BEGIN
OPEN C1;
FETCH C1 INTO Z;
WHILE (C1%FOUND) LOOP
DBMS_OUTPUT.PUT_LINE(Z.SID || '  ' || Z.SNAME);
FETCH C1 INTO Z;
END LOOP;
CLOSE C1;
END;
/
```

Suppose we want to display all sailors' information according to their rating with proper heading. (eg: 'Sailors with rating 1'….. etc) then we can write program as follows.
Ex:- Display records according to rating with proper heading.

```
DECLARE
I INTEGER:=1;
CURSOR C1 IS
SELECT * FROM SAILORS ORDER BY RATING;
Z C1%ROWTYPE;
BEGIN WHILE(I<=10)LOOP
DBMS_OUTPUT.PUT_LINE('SAILORS WITH RATING'||I);
FOR Z IN C1 LOOP
IF(Z.RATING=I)THEN
DBMS_OUTPUT.PUT_LINE(Z.SID||'  '||Z.SNAME||'  '||Z.AGE||'  '||Z.RATING);
END IF;
END LOOP;
I:=I+1;
END LOOP;
END;
/
```

Multiple cursors in a program :- We can use multiple cursors in a program.
Ex:- To display details of particular table sailors, boats, reserves according to users choice.

```
DECLARE

INPUT VARCHAR2(30):= '&INPUT';
CURSOR C1 IS SELECT * FROM SAILORS;
CURSOR C2 IS SELECT * FROM BOATS;
CURSOR C3 IS SELECT * FROM RESERVES;
BEGIN
IF(INPUT='SAILORS) THEN
DBMS_OUTPUT.PUT_LINE('SAILORS INFORMATION:');
FOR Z IN C1 LOOP
DBMS_OUTPUT.PUT_LINE(Z.SID||' '||Z.SNAME||' '||Z.AGE||' '||Z.RATING);
END LOOP;
ELSIF(INPUT='BOATS')THEN
DBMS_OUTPUT.PUT_LINE('BOATS INFORMATION:');
FOR X IN C2 LOOP
DBMS_OUTPUT.PUT_LINE(X.BID||' '||X.BNAME||'  '||X.COLOR);
END LOOP;
ELSIF(INPUT='RESERVES')THEN
DBMS_OUTPUT.PUT_LINE('RESERVES INFORMATION:');
```

```
FOR Y IN C3 LOOP
DBMS_OUTPUT.PUT_LINE(Y.SID||'  '||Y.BID||'  '||Y.DAY);
END LOOP;
ELSE
DBMS_OUTPUT.PUT_LINE('NO SUCH TABLE EXISTS');
END IF;
END;
/
```

Updating the Records :- Similar to inserting the values as well as selecting the values we can use the PL/SQL programming for updating the records in the given table.

Ex:- To update rating of sailors by 2 if rating is less than 5, by 1 if rating is >5 and doesn't change the rating if it is equal to 10.

```
DECLARE
CURSOR C1 IS SELECT * FROM SAILORS;
Z  C1%ROWTYPE;
BEGIN
FOR Z IN C1 LOOP
IF (Z.RATING<5) THEN
UPDATE SAILORS SET RATING=RATING+2 WHERE SID=Z.SID;
ELSIF (Z.RATING>5 AND Z.RATING<10) THEN
UPDATE SAILORS SET RATING=RATING+1 WHERE SID=Z.SID;
END IF;
END LOOP;
FOR Z IN C1 LOOP
DBMS_OUTPUT.PUT_LINE (Z.SID||'  '||Z.RATING);
END LOOP;
END;
/
```

Deleting the Records :- Similar to inserting and updating the values as well as selecting the values we can use the PL/SQL programming for deleting the records from the given table.

Ex:- Write a program to delete records from sailors table by reading SID from Keyboard.

```
DECLARE
BEGIN
DELETE FROM SAILORS WHERE SID='&SID';
END;
/
```

Passing parameters to Cursor

We can pass parameters to cursor. When you open cursor the value is passed to cursor and processing can be done there in cursor definition.

Ex:- suppose we want to display all sailors information according to their rating with proper heading. (eg: 'Sailors with rating 1'….. etc). Already we have seen same program with parameters to the cursor.
Following program illustrates how we can pass parameters to the cursor.

```
--Assume file name Cur_Par
DECLARE
CURSOR C1(R NUMBER) IS SELECT * FROM SAILORS WHERE RATING=R;
I INTEGER;
BEGIN
FOR I IN 1..10 LOOP
DBMS_OUTPUT.PUT_LINE('SAILORS WITH RATING '|| I || ' ARE');
DBMS_OUTPUT.PUT_LINE('SID   NAME        AGE');
FOR Z IN C1(I) LOOP
/* Its not compulsory to define variable using rowtype for simple cursor as well as for update
cursor */
DBMS_OUTPUT.PUT_LINE(Z.SID ||' ' ||Z.SNAME ||' '||Z.AGE);
END LOOP;
END LOOP;
END;
/
```

Output

```
SQL> @Cur_Par
SAILORS WITH RATING 1 ARE
4 Hemant 18 1
SAILORS WITH RATING 2 ARE
5 Rajendra 30 2
SAILORS WITH RATING 3 ARE
6 Satish 20 3
SAILORS WITH RATING 4 ARE
7 Shrikant 21 4
SAILORS WITH RATING 5 ARE
8 Shabaz 19 5
10 Kaushal 20 5
SAILORS WITH RATING 6 ARE
12 RAJ 19 6
SAILORS WITH RATING 7 ARE
1 aravind 19 7
SAILORS WITH RATING 8 ARE
9 Guru 36 8
SAILORS WITH RATING 9 ARE
3 Vijay 32 9
SAILORS WITH RATING 10 ARE
2 Amar 87 10
11 SUNNY 20 10
```

PL/SQL procedure successfully completed.

Use of Cursor for Update

(FOR UPDATE CURSOR, WHERE CURRENT of clause and CURSOR variable is used).

We can use update cursors for update operation only. The following example shows change of rating (Already we have written this program without using Update Cursor) using Update Cursor.

```
DECLARE
CURSOR C1 IS SELECT * FROM SAILORS FOR UPDATE;
BEGIN
FOR Z IN C1 LOOP
/* Its not compulsory to define variable using rowtype for update cursor as well as for simple
cursors */
IF(Z.RATING <=5) THEN
UPDATE SAILORS SET RATING= RATING+2 WHERE CURRENT OF C1;
ELSIF(Z.RATING>5 AND Z.RATING<10)
UPDATE SAILORS SET RATING= RATING+1 WHERE CURRENT OF C1;
END IF;
END LOOP;
END;
/
```

Error Handling using RAISE_APPLICATION_ERROR

Procedure RAISE_APPLICATION_ERROR is used to generate user-defined errors in the PL/SQL. The general syntax is

RAISE_APPLICATION_ERROR(ErrorCode, Error_Message [, TRUE/FALSE]);

The valid Error_Code is in range from –20000 to –20999.

The Error_Message length is maximum 2048 bytes.

The optional third parameter TRUE indicates that error message is put in stack. If FALSE is mentioned then error replaces all previous errors.

Example to illustrate RAISE_APPLICATION_ERROR

```
-- Assume file name Raise_Application_Error
DECLARE
A INTEGER:=&A;
B INTEGER:=&B;
C INTEGER;
BEGIN
IF(B=0)THEN
RAISE_APPLICATION_ERROR(-20001,'DIVISION BY ZERO');
ELSE
```

```
C:=A/B;
DBMS_OUTPUT.PUT_LINE('RESULT IS :'||C);
END IF;
END;
/
```

Output

```
SQL> @Raise_Application_Error
ENTER VALUE FOR A: 12
OLD   2: A INTEGER:=&A;
NEW   2:  A INTEGER:=12;
ENTER VALUE FOR B: 2
OLD  3: B INTEGER:=&B;
NEW  3: B INTEGER:=2;
RESULT IS :6
PL/SQL procedure successfully completed.

SQL> @Raise_Application_Error
ENTER VALUE FOR A: 15
OLD   2: A INTEGER:=&A;
NEW   2:  A INTEGER:=15;
ENTER VALUE FOR B: 0
OLD  3: B INTEGER:=&B;
NEW  3: B INTEGER:=0;
DECLARE
*
ERROR at line 1:
ORA-20001: DIVISION BY ZERO
ORA-06512: at line 8
```

Use of Commit, Savepoint & Rollback in PL/SQL

We can use these commands in PL/SQL. We can use any (Insert, Delete or Update) operations for Savepoint & Rollback.

The following program inserts a record into Sailors table then updates a record before we Commit a Savepoint is defined and we can use Rollback to undo the operations we have done after the Savepoint (i.e. deleting a Sailors record is undone). We can define number of Savepoint statements in a program and Rollback to any point.

```
BEGIN
 INSERT INTO SAILORS VALUES('32','HEMANT',10, 30);
 UPDATE SAILORS SET SNAME='RAJENDRA' WHERE SID='10';
 SAVEPOINT S1;
 DELETE FROM SAILORS WHERE SID='11';
```

```
ROLLBACK TO S1;
 COMMIT;
 END;
/
```

(You can even read these values from Keyboard)

Procedure in PL/SQL

Procedures are written for doing specific tasks. The general syntax of procedure is

CREATE OR REPLACE PROCEDURE <Pro_Name> (Par_Name1 [IN/OUT/ IN OUT] Par_Type1, ….) IS (Or we can write AS)
Local declarations;
BEGIN
PL/SQL Executable statements;
……. EXCEPTION
Exception Handlers;
END <Pro_Name>;

Mode of parameters

1) **IN Mode :-** IN mode is used to pass a value to Procedure/Function. Inside the procedure/function, IN acts as a constant and any attempt to change its value causes compilation error.
2) **OUT Mode :** The OUT parameter is used to return value to the calling routine. Any attempt to refer to the value of this parameter results in null value.
3) **IN OUT Mode :** IN OUT parameter is used to pass a value to a subprogram and for getting the updated value from the subprogram.

- **For writing Procedures we can directly type at SQL prompt or create a file.**
  SQL> ed File_Name
- **Type & save procedure.**
- **To create Procedure (before calling from other program.)**
  SQL> @File_Name
- **To use/call procedure, write a PL/SQL code and include call in the code using**
  Pro_Name(Par_List);

  Or you can execute from SQL Prompt as
  SQL>execute Pro_Name(Par_List)

For dropping Procedure/Function (Function described next)
SQL>DROP PROCEDURE Pro_Name;
SQL>DROP FUNCTION Fun_Name;

Examples
1) Simple program to illustrate Procedure.

```
-- Assume file name P1
CREATE OR REPLACE PROCEDURE P1(A NUMBER) AS
BEGIN
DBMS_OUTPUT.PUT_LINE('A:'||A);
END P1;
/
```

Now write PL/SQL code to use procedure in separate file.

```
-- Assume file name testP1
DECLARE
BEGIN
P1(100);
END;
/
```

Output

```
SQL> @P1
Procedure created.
SQL>@testP1
A:100
PL/SQL procedure successfully completed.
```

2) Program to illustrate Procedure with IN mode parameter.

```
-- Assume file name P2
CREATE OR REPLACE PROCEDURE P2(A IN NUMBER) AS
BEGIN
DBMS_OUTPUT.PUT_LINE('A:'||A);
END P2;
/

-- Assume file name testP2
DECLARE
X NUMBER;
BEGIN
X:=10;
DBMS_OUTPUT.PUT_LINE('X:'||X);
P2(X);
DBMS_OUTPUT.PUT_LINE('X:'||X);
END;
/
```

Output

```
SQL> @P2
Procedure created.
SQL>@testP2
X:10
A:10
```

X:10
PL/SQL procedure successfully completed.

3) Program to illustrate Procedure with OUT mode parameter.

```
-- Assume file name P3
CREATE OR REPLACE PROCEDURE P3(A OUT NUMBER) AS
 BEGIN
 A:=100;
 DBMS_OUTPUT.PUT_LINE('A:'|| A);
 END P3;
 /

-- Assume file name testP3
DECLARE
X NUMBER;
BEGIN
X:=50;
DBMS_OUTPUT.PUT_LINE('X:'||X);
P3(X);
DBMS_OUTPUT.PUT_LINE('X:'||X);
END;
/
```

Output

```
SQL> @P3
Procedure created.
SQL>@testP3
X:50
A:100
X:100
PL/SQL procedure successfully completed.
```

4) Program to illustrate Procedure with OUT mode parameter.

```
-- Assume file name P4
CREATE OR REPLACE PROCEDURE P4(A OUT NUMBER) AS
BEGIN
DBMS_OUTPUT.PUT_LINE('A:'||A);
END P4;
/

-- Assume file name testP4
DECLARE
X NUMBER;
BEGIN
X:=10;
DBMS_OUTPUT.PUT_LINE('X:'||X);
```

```
P4(X);
DBMS_OUTPUT.PUT_LINE('X:'||X);
END;
/
```

Output

```
SQL> @P4
Procedure created.
SQL>@testP4
X:10
A:
X:
PL/SQL procedure successfully completed.
```

5) Program to illustrate Procedure with IN OUT mode parameter.

```
--Assume file name P5
CREATE OR REPLACE PROCEDURE P5(A IN OUT NUMBER) AS
BEGIN
DBMS_OUTPUT.PUT_LINE('A:' || A);
END P5;
/

-- Assume file name testP5
DECLARE
X NUMBER;
BEGIN
X:=10;
DBMS_OUTPUT.PUT_LINE('X:'|| X);
P5(X);
DBMS_OUTPUT.PUT_LINE('X:'|| X);
END;
/
```

Output

```
SQL> @P5
Procedure created.
SQL>@testP5
X:10
A:10
X:10
PL/SQL procedure successfully completed.
```

Functions in PL/SQL

Similar to Procedure we can create Functions which can return one value to the calling program. The syntax of function is

CREATE OR REPLACE FUNCTION <Fun_Name> (Par_Name1 [IN/OUT/ IN OUT] Par_Type1, ….)
RETURN return_datatype IS
Local declarations;
BEGIN
PL/SQL Executable statements;
……. EXCEPTION
Exception Handlers;
END <Fun_Name>;

- For writing Function we can directly type at SQL prompt or create a file.
  SQL> ed File_Name
- Type & save Function.
- To create Function (before calling from other program.)

SQL> @File_Name

- To use/call Function we have two ways.
  1) Write a PL/SQL code and include call in the code using
     x=Fun_Name(Par_List);
  2) You can execute from SQL Prompt as a select query
     SQL>Select Fun_Name(Par_List) from Dual;

Ex :- Program to illustrate Function. (Finding Square of a number)

```
-- Assume file name Fun
CREATE OR REPLACE FUNCTION FUN(A NUMBER)
RETURN NUMBER IS
BEGIN
RETURN (A*A);
END FUN;
/

-- Assume file name testFun
DECLARE
X NUMBER:=&X;
S NUMBER;
BEGIN
S:=FUN(X);
DBMS_OUTPUT.PUT_LINE('SQUARE OF A NUMBER'|| S);
END;
/
```

Output

SQL> @Fun

```
Function created.
SQL> @testFun
ENTER VALUE FOR X: 10
OLD  2: X NUMBER:=&X;
NEW  2: X NUMBER:=10;
SQUARE OF A NUMBER100
PL/SQL procedure successfully completed.
```