Experiment-1: Queries for Creating, Altering and Dropping Tables, Views and Constraints.

DDL Commands: DDL stands for Data Definition Language. Here we find CREATE, ALTER, DROP and TRUNCATE commands.

1) CREATE TABLE: It is used to create a new table definition. Its syntax is:

```
CREATE TABLE <table_name>
(
<column_name1><datatype>(<size>)[<constraint1>]
<column_name2><datatype>(<size>)[<constraint1>]
[<constraint-list>]
);
```

Creating a table

```
SQL> create table students
  2  (
  3  rollno varchar2(30),
  4  name varchar2(30),
  5  branch varchar2(15)
  6  );

Table created.
```

2) DROP TABLE COMMAND: It is used to delete all the existing records from the table and the definition also.

Syntax: DROP TABLE <table name>;

Dropping table

```
SQL> drop table stu_details;

Table dropped.

SQL> select * from tab;

TNAME                          TABTYPE  CLUSTERID
------------------------------ ------- ----------
STUDENT1                        TABLE
STUDENTS2                       TABLE
STU                            TABLE
STU1                           TABLE
STU2                           TABLE
STUD                           TABLE
STU3                           TABLE
STU4                           TABLE
STU5                           TABLE
BIN$QdFqkisyT1CzNapxV9Kqpw==$0 TABLE
BIN$OYWM4Kc6S3+WIMslmtnp/w==$0 TABLE

11 rows selected.
```

To View Schema

```
SQL> desc students
 Name                           Null?    Type
```

ROLLNO	VARCHAR2(30)
NAME	VARCHAR2(30)
BRANCH	VARCHAR2(15)

3) ALTER TABLE COMMAND:

Altering Table: The alter table command is used to modify the structure of existing table. (i.e adding a column, add an integrity constraint etc.).

Adding Columns: The new column will be added with NULL values for all rows currently in table.

Syntax: alter table <table_name>
add (<col1><datatype><size> [<constraint>],
<col2><datatype><size><constraint>],…….);

Modifying Column Definitions: To change datatype, size, default value and NOT NULL column constraint of a column definition.

Syntax: alter table <table_name>
modify (<col_name><new_datatype><new_size>);

Dropping Columns: The column can be deleted as follows.

Syntax: alter table <table_name>
drop (<col1>,<col2>,…….);

To change column name:

Syntax: ALTER TABLE <table name> RENAME COLUMN <old column name> TO <new column name>;

Altering the table

```
SQL> alter table students
  2  add age integer;

Table altered.

SQL> alter table students
  2  drop column branch;

Table altered.

SQL> desc students
```

Name	Null?	Type
ROLLNO		VARCHAR2(30)
NAME		VARCHAR2(30)
AGE		NUMBER(38)

4) Rename: RENAME allows you to rename an existing table

Syntax: RENAME old-table-name TO new-table-name

Rename a table

```
SQL> rename students to stu_details2;

Table renamed.
```

5) Truncate: It is used to delete complete data from an existing table

Syntax: Truncate table tablename;

Truncate

```
SQL> truncate table students;

Table truncated.
```

Constraints:

Not null

```
SQL> create table stu
  2  (
  3  rno integer not null,
  4  name varchar2(20)
  5  );

Table created.
SQL> insert into stu values(501,'rani');

1 row created.
SQL> insert into stu values(501,'rani');

1 row created.
SQL> insert into stu values(null,'kamala');
insert into stu values(null,'kamala')
                       *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("CSE19517"."STU"."RNO")
```

Unique

```
SQL> create table stu1
  2  (
  3  rno integer,
  4  name varchar2(20),
  5  unique(rno)
  6  );
Table created.

SQL> insert into stu1 values(501,'rani');

1 row created.

SQL> insert into stu1 values(501,'rani');
insert into stu1 values(501,'rani')
*
ERROR at line 1:
ORA-00001: unique constraint (CSE19517.SYS_C005728) violated

SQL> insert into stu1 values(null,'kamala');

1 row created.
```

Primary key

```
SQL> create table stu2
(
rno integer,
name varchar2(20),
primary key(rno)
);
Table created.

SQL> insert into stu2 values(501,'rani');

1 row created.

SQL> insert into stu2 values(501,'rani');
insert into stu2 values(501,'rani')
*
ERROR at line 1:
ORA-00001: unique constraint (CSE19517.SYS_C005730) violated

SQL> insert into stu2 values(null,'kamala');
insert into stu2 values(null,'kamala')
                 *
ERROR at line 1:
ORA-01400: cannot insert NULL into ("CSE19517"."STU2"."RNO")

SQL> create table stu5
  2  (
  3  rno integer,
  4  name varchar2(20)
  5  );

Table created.

SQL> alter table stu5
  2  add primary key(rno);

Table altered.

SQL> desc stu5
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 RNO                                       NOT NULL NUMBER(38)
 NAME                                               VARCHAR2(20)
```

Foreign key

```
SQL> create table stud
  2  (
  3  rno integer,
  4  fee integer,
```

```
  5  foreign key(rno) references stu2(rno)
  6  );

Table created.

SQL> insert into stud values(501,6000);

1 row created.

SQL> insert into stud values(502,8000);
insert into stud values(502,8000)
*
ERROR at line 1:
ORA-02291: integrity constraint (CSE19517.SYS_C005731) violated - parent key
not found

SQL> delete from stud where rno=501;

1 row deleted.

SQL> delete from stu2 where rno=501;

1 row deleted.
```

Default

```
SQL> create table stu3
  2  (
  3  rno integer,
  4  name varchar2(20),
  5  gender char(1) default 'F'
  6  );

Table created.

SQL> insert into stu3 values(501,'ravi','M');

1 row created.

SQL> insert into stu3 values(501,'suma',default);

1 row created.
```

Check

```
SQL> create table stu4
  2  (
  3  rno integer,
  4  name varchar2(20),
  5  marks integer,
  6  check (marks<101)
  7  );
```

Table created.

```
SQL> insert into stu4 values(501,'ravi',101);
insert into stu4 values(501,'ravi',101)
*
ERROR at line 1:
ORA-02290: check constraint (CSE19517.SYS_C005733) violated
```

On delete Cascade:

ON DELETE CASCADE clause in MySQL is used to automatically remove the matching records from the child table when we delete the rows from the parent table. It is a kind of referential action related to the foreign key.

Suppose we have created two tables with a FOREIGN KEY in a foreign key relationship, making both tables a parent and child. Next, we define an ON DELETE CASCADE clause for one FOREIGN KEY that must be set for the other to succeed in the cascading operations. If the ON DELETE CASCADE is defined for one FOREIGN KEY clause only, then cascading operations will throw an error.

```
SQL> create table t1(
2  t1id integer,
3  t1desc varchar2(30),
4  primary key(t1id));
Table created.

SQL> create table t2(
2  t1t2id integer,
3  t2id integer,
4  t2desc varchar2(30),
5  primary key(t2id),
6  foreign key(t1t2id) references t1(t1id) on delete cascade);
Table created.

SQL> create table t3(
2  t2t3id integer,
3  t3id integer,
4  t3desc varchar2(30),
5  primary key(t3id),
6  foreign key(t2t3id) references t2(t2id) on delete cascade);
Table created.

SQL> insert into t1 values(1,'t1 one');
1 row created.
SQL> insert into t2 values(1,1,'t2 one');
1 row created.
SQL> insert into t3 values(1,1,'t3 one');
1 row created.
SQL> delete from t1 cascade;
1 row deleted.
SQL> delete from t2 cascade;
0 rows deleted.
SQL> drop table t3;
Table dropped.
SQL> drop table t2;
Table dropped.
SQL> drop table t1;
Table dropped.
```

On delete Set Null:

A foreign key with "set null on delete" means that if a record in the parent table is deleted, then the corresponding records in the child table will have the foreign key fields set to NULL. The records in the child table will not be deleted in SQL Server.

A foreign key with set null on delete can be created using either a CREATE TABLE statement or an ALTER TABLE statement.

```
SQL> create table t1(
2  t1id integer,
3  t1desc varchar2(30),
4  primary key(t1id));
Table created.

SQL> create table t2(
2  t2id integer,
3  t1t2id integer,
4  t2desc varchar2(30),
5  primary key(t2id),
6  foreign key(t1t2id) references t1(t1id) on delete set null);
Table created.

SQL> create table t3(
2  t2t3id integer,
3  t3id integer,
4  t3desc varchar2(30),
5  primary key(t3id),
6  foreign key(t2t3id) references t2(t2id) on delete set null);
Table created.

SQL> insert into t1 values(1,'t1 one');
1 row created.
SQL> insert into t2 values(1,1,'t2 one');
1 row created.
SQL> insert into t3 values(1,1,'t3 one');
1 row created.
SQL> delete from t1 where t1id=1;
1 row deleted.
SQL> select * from t1;
no rows selected

SQL> select * from t2;
     T2ID     T1T2ID T2DESC
---------- ---------- ------------------------------
         1          t2 one

SQL> select * from t3;
   T2T3ID       T3ID T3DESC
---------- ---------- ------------------------------
         1          1 t3 one
```

Experiment-2: Queries to Retrieve and Change Data: Select, Insert, Delete and Update.

DML COMMANDS are INSERT, UPDATE, DELETE and SELECT.

Creating table

```
SQL> create table students
  2  (
  3  rollno varchar2(30),
  4  name varchar2(30)
  5  );
```

Table created.

INSERT COMMAND: This command is used to create data into the table which is already defined through DDL commands. The data can be entered in the form of rows and columns.

Syntax: INSERT INTO <Table name>
(column1, [column2, …….,columnN])
values (column1value,column2value,…..,columnNvalue);
OR
INSERT INTO <Table name>
Values (column1value,column2value,…..,columnNvalue);

Inserting Data into the table

```
SQL> insert into students values('19A91A0501','Ravi');
```

1 row created.

```
SQL> insert into students values('19A91A0502','Suma');
```

1 row created.

SELECT COMMAND: This command is used to view a single row or multiple rows or single column or multiple columns of a table.

Syntax:
SELECT *|{[DISTINCT] column|expression [alias],...}
FROM table [where<condition>];

Displaying Data from the table

```
SQL> select * from students;

ROLLNO                         NAME
------------------------------ ------------------------------
19A91A0501                      Ravi
19A91A0502                      Suma

SQL> select name from students;

NAME
------------------------------
Ravi
```

Suma

SQL> select * from students where rollno='19A91A0501';

ROLLNO	NAME
19A91A0501	Ravi

DELETE COMMAND: This command is used to remove a single row or multiple rows of a table.
Syntax: DELETE FROM<Table_name>[where<condition>];

Deleting a row from the table
SQL> delete from students where rollno='19A91A0501';

1 row deleted.

UPDATE COMMAND: This command is used to modify or change or replace the existingdata of a table.

Syntax:
```
UPDATE <Table_name>
Set <column1>=<column1value>
[,<column2>=<column2value>,……
,<columnN>=<columnNvalue>]
[where<condition>];
```

Updating a row in the table
```
SQL> update students
  2  set name='Rose'
  3  where rollno='19A91A0502';

1 row updated.
SQL> insert into students values(&rollno,'&name');
Enter value for rollno: 501
Enter value for name: sai
old   1: insert into students values(&rollno,'&name')
new   1: insert into students values(501,'sai')

1 row created.

SQL> /
Enter value for rollno: 517
Enter value for name: sree
old   1: insert into students values(&rollno,'&name')
new   1: insert into students values(517,'sree')

1 row created.
```

```
SQL> insert into students(name,rollno) values('padma',503);

1 row created.

SQL> select * from students;

ROLLNO                         NAME
------------------------------ ------------------------------
19A91A0502                      Rose
501                    sai
517                    sree
503                    padma
```

Experiment-3:

3.1) Queries to facilitate acquaintance of Built-in Functions: String Functions, Numeric Functions, Date Functions and Conversion Functions.

String Functions

CONCAT(): The CONCAT() function adds two or more strings together.
Syntax: CONCAT(*string1*, *string2*, ..., *string_n*)

```
SQL> select concat('aditya','engg') from dual;
CONCAT('AD
----------
adityaengg

SQL> select concat(concat('aditya','engg'),'college') from dual;
CONCAT(CONCAT('AD
-----------------
adityaenggcollege
```

The concatenate operator (||): Sometimes we want to combine values from different columns, either in the WHERE clause or for the results. SQLite uses double-pipes (||) - more formally known as the concatenation operator - to combine strings. You can combine both literal strings (in quotation marks) and column values using this operator.

```
SQL> select 'aditya'||'engg' from dual;
'ADITYA'||
----------
adityaengg
```

LPAD() Function: The LPAD() function left-pads a string with another string, to a certain length.
Syntax: LPAD(*string*, *length*, *lpad_string*)

```
SQL> select lpad('aditya',15,'*') as lpad from dual;
LPAD
---------------
**********aditya
```

RPAD() Function: The RPAD() function right-pads a string with another string, to a certain length.
Syntax: RPAD(*string*, *length*, *rpad_string*)

```
SQL> select rpad('aditya',15,'*') as rpad from dual;
RPAD
---------------
aditya*********
```

LTRIM() Function: The LTRIM() function removes leading spaces from a string. If we want to specify the characters to be removed or if we want to remove a substring from the original or given string, then we can also achieve that by using the LTRIM().
Syntax: LTRIM(*string*)

```
SQL> select ltrim('123123123rama123','123') from dual;
LTRIM('
```

---
rama123

RTRIM() Function: The RTRIM() function removes trailing spaces from a string. This string function truncates the given character or sub-string from the right of the given original string.
Syntax: RTRIM(*string*)
SQL> select rtrim('123123123rama123','123') from dual;
RTRIM('123123

123123123rama

UPPER() Function: The UPPER() function converts a string to upper-case.
Syntax: UPPER(*text*)
SQL> select upper('aditya') from dual;
UPPER(

ADITYA

LOWER() Function: The LOWER() function converts a string to lower-case.
Syntax: LOWER(*text*)
SQL> select lower('ADITYA') from dual;
LOWER(
---
aditya

LENGTH() Function: The LENGTH() function returns the length of a string (in bytes).
Syntax: LENGTH(*string*)
SQL> select length('aditya') from dual;
LENGTH('ADITYA')

6

SUBSTR() Function: The SUBSTR() function extracts a substring from a string (starting at any position).
Syntax: SUBSTR(*string*, *start*, *length*)
SQL> select substr('abcdefg',-3,2) from dual;
SU
--
ef

INSTR() Function: The INSTR() function returns the position of the first occurrence of a string in another string.
This function performs a case-insensitive search.
Syntax: INSTR(*string1*, *string2*,*position*)
SQL> select instr('abab','b') from dual;
INSTR('ABAB','B')

2

SQL> select instr('abab','b',3) from dual;
INSTR('ABAB','B',3)

4

ASCII() Function: The ASCII() function returns the ASCII value for the specific character.
Syntax: ASCII(*character*)
SQL> select ascii('A') from dual;
ASCII('A')

65

CHR() Function: The SQL CHR function accepts an ASCII code and returns the corresponding character. The CHR function is the opposite of the ASCII function.
Syntax: CHR(ascii)
SQL> select chr(97) from dual;
C
-
a

REVERSE() Function: The REVERSE() function reverses a string and returns the result.
Syntax: REVERSE(*string*)
SQL> select reverse('aditya') from dual;
REVERS
---
aytida

INITCAP() Function: It is a string function that changes the first letter of a string to uppercase. The remaining letters are made lowercase.
Syntax: INITCAP(string)
SQL> select initcap('adityaengg') from dual;
INITCAP('A

Adityaengg

Numeric Functions

ABS() Function: The ABS() function returns the absolute (positive) value of a number.
Syntax: ABS(*number*)
SQL> select abs(19) from dual;
ABS(19)

19
SQL> select abs(-19) from dual;
ABS(-19)

19

SIGN() Function: The SIGN() function returns the sign of a number.
This function will return one of the following:
If number > 0, it returns 1; If number = 0, it returns 0; If number < 0, it returns -1

Syntax: SIGN(*number*)
SQL> select sign(19) from dual;
SIGN(19)

1
SQL> select sign(-19) from dual;
SIGN(-19)

-1

POWER() Function: The POWER() function returns the value of a number raised to the power of another number.
Syntax: POWER(*x*, *y*)
Where x → base and y → exponent.
SQL> select power(3,2) from dual;
POWER(3,2)

9

SQRT() Function: The SQRT() function returns the square root of a number.
Syntax: SQRT(*number*)
SQL> select sqrt(9) from dual;
SQRT(9)

3

CEIL() Function: The CEIL() function returns the smallest integer value that is bigger than or equal to a number.
Syntax: CEIL(*number*)
SQL> select ceil(2.2) from dual;
CEIL(2.2)

3
SQL> select ceil(-2.2) from dual;
CEIL(-2.2)

-2

FLOOR() Function: The FLOOR() function returns the largest integer value that is smaller than or equal to a number.
Syntax: FLOOR(*number*)
SQL> select floor(2.2) from dual;
FLOOR(2.2)

2
SQL> select floor(-2.2) from dual;
FLOOR(-2.2)

-3

MOD() Function: The MOD() function returns the remainder of a number divided by another number.
Syntax: MOD(*x*, *y*)

```
SQL> select mod(150,7) from dual;
MOD(150,7)
----------
         3
```

ROUND() Function: The ROUND() function rounds a number to a specified number of decimal places.
Syntax: ROUND(*number*, *decimals*)

```
SQL> select round(66.666,2) from dual;
ROUND(66.666,2)
---------------
          66.67
```

TRUNC() Function: returns n1 truncated to n2 decimal places. If n2 is omitted, then n1 is truncated to 0 places. n2 can be negative to truncate (make zero) n2 digits left of the decimal point.
Syntax: TRUNC(*number*,*decimals*)

```
SQL> select trunc(66.666,2) from dual;
TRUNC(66.666,2)
---------------
          66.66
```

EXP() Function: The EXP() function returns *e* raised to the power of the specified number.
Syntax: EXP(*number*)

```
SQL> select exp(3) from dual;
   EXP(3)
----------
20.0855369
```

LOG() Function: The LOG() function returns the natural logarithm of a specified *number*, or the logarithm of the *number* to the specified *base*.
Syntax: LOG(*base*, *number*)

```
SQL> select log(2,2) from dual;
  LOG(2,2)
----------
         1
```

Date Functions

SYSDATE() Function: The SYSDATE() function returns the current date and time.
Syntax: SYSDATE()
Note: The date and time is returned as "YYYY-MM-DD HH:MM:SS" (string) or as YYYYMMDDHHMMSS (numeric).

```
SQL> select sysdate from dual;
SYSDATE
---------
22-FEB-23
```

```
SQL> select sysdate+1 from dual;
SYSDATE+1
---------
23-FEB-23

SQL> select sysdate-1 from dual;
SYSDATE-1
---------
21-FEB-23
```

EXTRACT() Function: The EXTRACT() function extracts a part from a given date.
Syntax: EXTRACT(*part* FROM *date*)

```
SQL> select extract(year from sysdate) from dual;
EXTRACT(YEARFROMSYSDATE)
------------------------
                    2023

SQL> select extract(month from sysdate) from dual;
EXTRACT(MONTHFROMSYSDATE)
-------------------------
                       2

SQL> select extract(day from sysdate) from dual;
EXTRACT(DAYFROMSYSDATE)
-----------------------
                     22
```

TO_CHAR() Function: TO_CHAR function is used to typecast a numeric or date input to character type with a format model (optional).
Syntax: TO_CHAR(date, 'format_model')

```
SQL> select to_char(sysdate,'yyyy
/mm/dd') from dual;
TO_CHAR(SY
----------
2023/02/22

SQL> select to_char(sysdate,'hh:mm:ss') from dual;
TO_CHAR(
--------
01:02:49
```

ADD_DATE() Function: The ADD_MONTHS function accepts two parameters which are the initial date and the number of months to be added to it. The ADD_MONTHS function returns a value of the date data type.
Syntax: ADD_MONTHS(init_date, add_months)

```
SQL> select add_months(sysdate,2) from dual;
ADD_MONTH
---------
22-APR-23
```

NEXT_DAY() Function: It is used to return the first weekday that is greater than the given date. So this function will take the input from the user that is the date and the weekday and then it will return the date which is greater than the given date according to the weekday.

Syntax: NEXT_DAY(DATE,WEEKDAY)

```
SQL> select next_day(sysdate,'wednesday') from dual;
NEXT_DAY(
---------
01-MAR-23

SQL> select next_day('10-dec-2019','tuesday') from dual;
NEXT_DAY(
---------
17-DEC-19
```

LAST_DAY() Function: The LAST_DAY() function in MySQL can be used to know the last day of the month for a given date or a datetime. The LAST_DAY() function takes a *date* value as argument and returns the last day of month in that *date*. The *date* argument represents a valid date or datetime.

Syntax: LAST_DAY(Date);

```
SQL> select last_day(sysdate) from dual;
LAST_DAY(
---------
28-FEB-23
```

MONTHS_BETWEEN() Function: Returns number of months between date1 and date2. If *date1* is earlier than *date2*, then the result is negative. If *date1* and *date2* are either the same days of the month or both last days of months, then the result is always an integer.

Syntax: MONTHS_BETWEEN(date1, date2)

```
SQL> select months_between(to_date('09-dec-2020','dd-mm-yyyy'),to_date('09-dec-
2019','dd-mm-yyyy')) from dual;
MONTHS_BETWEEN(TO_DATE('09-DEC-2020','DD-MM-YYYY'),TO_DATE('09-DEC-
2019','DD-MM-
----------------------------------------------------------------------------
```

3.2) Queries using operators in SQL.

Arithmetic Operators

The **Arithmetic Operators** perform the mathematical operation on the numerical data of the SQL tables. These operators perform addition, subtraction, multiplication, and division operations on the numerical operands.

Addition Operator(+): The Addition Operator in SQL performs the addition on the numerical data of the database table. In SQL, we can easily add the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also add the numbers to the existing numbers of the specific column.

Syntax: SELECT operand1 + operand2;

```
SQL> select 4+5 as add from dual;
      ADD
----------
         9
```

Subtraction Operator(-): The Subtraction Operator in SQL performs the subtraction on the numerical data of the database table. In SQL, we can easily subtract the numerical values of two columns of the same table by specifying both the column names as the first and second operand. We can also subtract the number from the existing number of the specific table column.

Syntax: SELECT operand1 - operand2;

```
SQL> select 9-5 as sub from dual;
      SUB
----------
         4
```

Multiplication Operator(*):In SQL, we can easily multiply the numerical values of two columns of the same table by specifying both the column names as the first and second operand.

Syntax: SELECT operand1 * operand2;

```
SQL> select 2*3 as mul from dual;
      MUL
----------
         6
```

Division Operator(/): The Division Operator in SQL divides the operand on the left side by the operand on the right side.

Syntax: SELECT operand1 / operand2;

```
SQL> select 8/4 as div from dual;
      DIV
----------
         2
```

Modulus Operator(%): The Modulus Operator in SQL provides the remainder when the operand on the left side is divided by the operand on the right side.

Syntax: SELECT operand1 % operand2;

```
SQL> select mod(8,4) as mod from dual;
      MOD
----------
         0
```

Comparison Operators

The **Comparison Operators** in SQL compare two different data of SQL table and check whether they are the same, greater, and lesser. The SQL comparison operators are used with the WHERE clause in the SQL queries.

```
SQL> create table student
  2  (
  3  sid integer,
  4  sname varchar2(30),
  5  age integer
  6  );
Table created.

SQL> insert into student values(501,'Akash',21);
1 row created.

SQL> insert into student values(502,'Thanmayi',24);
1 row created.
```

Equal Operator(=): This operator is highly used in SQL queries. The **Equal Operator** in SQL shows only data that matches the specified value in the query.

```
SQL> select sid,sname from student where age=21;
       SID SNAME
---------- ------------------------------
       501 Akash
```

Greater Than Operator(>): The **Greater Than Operator** in SQL shows only those data which are greater than the value of the right-hand operand.

```
SQL> select sid,sname from student where age>10;
no rows selected
```

Less Than Operator(<): The **Less Than Operator** in SQL shows only those data from the database tables which are less than the value of the right-side operand.

```
SQL> select sid,sname from student where age<20;
no rows selected
```

Greater Than Equal to Operator(>=): The **Greater Than Equals to Operator** in SQL shows those data from the table which are greater than and equal to the value of the right-hand operand.

```
SQL> select sid,sname from student where age>=21;
       SID SNAME
---------- ------------------------------
       501 Akash
       502 Thanmayi
```

Less Than Equal To Operator(<=): The **Less Than Equals to Operator** in SQL shows those data from the table which are lesser and equal to the value of the right-side operand.

```
SQL> select sid,sname from student where age<=30;
       SID SNAME
---------- ------------------------------
       501 Akash
```

502 Thanmayi

Not Equal Operator(<>): The **Equal Not Operator** in SQL shows only those data that do not match the query's specified value. This operator returns those records or rows from the database views and tables if the value of both operands specified in the query is not matched with each other.

```
SQL> select sid,sname from student where age<>50;
      SID SNAME
---------- ------------------------------
      501 Akash
      502 Thanmayi
```

Logical Operators

The **Logical Operators** in SQL perform the Boolean operations, which give two results **True and False.** These operators provide **True** value if both operands match the logical condition.

AND Operator: The **AND operator** in SQL would show the record from the database table if all the conditions separated by the AND operator evaluated to True. It is also known as the conjunctive operator and is used with the WHERE clause.

```
SQL> select sid,sname from student where age>20 and age<30;
      SID SNAME
---------- ------------------------------
      501 Akash
      502 Thanmayi
```

OR Operator: The OR operator in SQL shows the record from the table if any of the conditions separated by the OR operator evaluates to True. It is also known as the conjunctive operator and is used with the WHERE clause.

```
SQL> select sid,sname from student where age>=21 or age<25;
      SID SNAME
---------- ------------------------------
      501 Akash
      502 Thanmayi
```

BETWEEN Operator: The **BETWEEN operator** in SQL shows the record within the range mentioned in the SQL query. This operator operates on the numbers, characters, and date/time operands.

If there is no value in the given range, then this operator shows NULL value.

```
SQL> select sid,sname from student where age between 20 and 25;
      SID SNAME
---------- ------------------------------
      501 Akash
      502 Thanmayi
```

NOT Operator: The **NOT operator** in SQL shows the record from the table if the condition evaluates to false. It is always used with the WHERE clause.

```
SQL> select sid,sname from student where not age=21;
      SID SNAME
---------- ------------------------------
      502 Thanmayi
```

Like Operators

The LIKE operator in SQL Server is used to search for character string with the specified pattern using wildcards in the column. In SQL Server, pattern means its specific string of characters with wildcards to search for matched expressions. Generally, we will use this LIKE operator in the WHERE clause.

```
SQL> create table customers
  2  (
  3  name varchar2(30),
  4  city varchar2(30)
  5  );
Table created.

SQL> insert into customers values('Ajay','Perry Ridge');
1 row created.

SQL> insert into customers values('Pavani','Downtown');
1 row created.

SQL> insert into customers values('Ravi','Paris');
1 row created.
```

Using '%' wildcard:

The following SQL query will return all cities with the name containing the word 'idge' anywhere in the name column because we mentioned patterns like '%idge%'. This means it will check for the respective word anywhere in the column irrespective of characters in front or back.

```
SQL> select * from customers where city like '%idge%';
NAME                           CITY
------------------------------ ------------------------------
Ajay                           Perry Ridge
```

'_' will represent a single character in the expression. As we are looking for a city name containing the word which starts with 'P' and followed by having four more characters.

```
SQL> select * from customers where city like 'P____';
NAME                           CITY
------------------------------ ------------------------------
Ravi                           Paris
```

Is Null and Is Not Null

```
SQL> insert into student values(503,'Ajay',NULL);
1 row created.

SQL> insert into student values(504,'Suma',NULL);
1 row created.
```

IS NULL Operator: The IS NULL operator is used to test for empty values (NULL values).

```
SQL> select sid,sname from student where age is null;
       SID SNAME
---------- ------------------------------
       503 Ajay
```

504 Suma

IS NOT NULL Operator: The IS NOT NULL operator is used to test for non-empty values (NOT NULL values).

SQL> select sid,sname from student where age is not null;

SID SNAME
---------- ------------------------------
501 Akash
502 Thanmayi

Set Operators

UNION: The UNION operator is used to combine the result-set of two or more SELECT statements.

- Every SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in every SELECT statement must also be in the same order

Syntax: SELECT *column_name(s)* FROM *table1*
UNION
SELECT *column_name(s)* FROM *table2*;

Find Sailors name who have reserves a red or a green boat using UNION?

SQL> select s.sname from sailors s,boats b,reserves r where s.sid=r.sid and b.bid=r.bid and b.color='Red' union select s1.sname from sailors s1,boats b1,reserves r1 where s1.sid=r1.sid and b1.bid=r1.bid and b1.color='Green';

SNAME

Dustin
Horatio
Lubber
Ravi

UNION ALL: The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL

Syntax: SELECT *column_name(s)* FROM *table1*
UNION ALL
SELECT *column_name(s)* FROM *table2*;

Find Sailors name who have reserved a red or a green boat using UNION ALL?

SQL> select s.sname from sailors s,boats b,reserves r where s.sid=r.sid and b.bid=r.bid and b.color='Red' union all select s1.sname from sailors s1,boats b1,reserves r1 where s1.sid=r1.sid and b1.bid=r1.bid and b1.color='Green';

SNAME

Dustin
Dustin
Lubber
Lubber
Horatio
Dustin
Lubber
Ravi

INTERSECT: To compare the rows of two or more Oracle SELECT statements, the Oracle INTERSECT operator is used. After the comparing process, the INTERSECT operator returns the common or intersecting records from the corresponding columns of the selected expressions.

There are however two mandatory conditions for using the INTERSECT operator in Oracle.

- Each SELECT statement must have the same number of expressions.
- Each corresponding expression in the different SELECT statement should be of the same data type.

Syntax: SELECT column1 , column2 .... FROM table_names WHERE condition
INTERSECT
SELECT column1 , column2 .... FROM table_names WHERE condition;

Find Sailors name who have reserved a red and a green boat using INTERSECT?

```
SQL> select s.sname from sailors s,boats b,reserves r where s.sid=r.sid and b.bid=r.bid and
b.color='Red' intersect select s1.sname from sailors s1,boats b1,reserves r1 where
s1.sid=r1.sid and b1.bid=r1.bid and b1.color='Green';
SNAME
----------
Dustin
Lubber
```

MINUS: The Minus Operator in SQL is used with two SELECT statements. The MINUS operator is used to subtract the result set obtained by first SELECT query from the result set obtained by second SELECT query.

Syntax: SELECT column1 , column2 , ... columnN FROM table_name WHERE condition
MINUS
SELECT column1 , column2 , ... columnN FROM table_name WHERE condition;

Find the Sailors name who have reserves a red boat but not a green boat?

```
SQL> select s.sname from sailors s,boats b,reserves r where s.sid=r.sid and b.bid=r.bid and
b.color='Red' minus select s1.sname from sailors s1,boats b1,reserves r1 where s1.sid=r1.sid
and b1.bid=r1.bid and b1.color='Green';
SNAME
----------
Horatio
```

Aggregate Operators

COUNT(): The COUNT() function returns the number of rows that matches a specified criterion.

Syntax: SELECT COUNT(*column_name*) FROM *table_name* WHERE *condition*;

Find number of Sailors?

```
SQL> select count(*) from sailors;
  COUNT(*)
----------
        10
```

Find count of distinct ratings of Sailors whose age is less than 40?

```
SQL> select count(distinct s.rating) from sailors s where s.age<40;
```

COUNT(DISTINCTS.RATING)

6

Find count of distinct sailors who reserved the boats and having age<40?
SQL> select count(distinct s.sid) from reserves r,sailors s where s.sid=r.sid and s.age<40;
COUNT(DISTINCTS.SID)

2

AVG(): The AVG() function returns the average value of a numeric column.
Syntax: SELECT AVG(*column_name*) FROM *table_name* WHERE *condition*;

Find the average of age of Sailors?
SQL> select avg(age) from sailors;
AVG(AGE)

36.85

Find the average of age of Sailors whose rating=10?
SQL> select avg(age) from sailors where rating=10;
AVG(AGE)

25.5

MIN(): The MIN() function returns the smallest value of the selected column.
Syntax: SELECT MIN(*column_name*) FROM *table_name* WHERE *condition*;

Find minimum age in Sailors?
SQL> select min(age) from sailors;
MIN(AGE)

16

MAX(): The MAX() function returns the largest value of the selected column.
Syntax: SELECT MAX(*column_name*) FROM *table_name* WHERE *condition*;

Find maximum age in Sailors?
SQL> select max(age) from sailors;
MAX(AGE)

63.5

SUM(): The SUM() function returns the total sum of a numeric column.
Syntax: SELECT SUM(*column_name*) FROM *table_name* WHERE *condition*;
Find sum of ratings in Sailors?
SQL> select sum(rating) from sailors;
SUM(RATING)

66

Experiment-4:

4.1) Queries using Group By, Order By, and Having Clauses.

```
SQL> create table company
  2  (
  3  compname varchar2(20),
  4  amount integer
  5  );
Table created.

SQL> select * from company;
COMPNAME              AMOUNT
-------------------- ----------
Wipro                  5000
IBM                    9000
Dell                  10000
Wipro                  4000
Dell                   6000
```

GROUP BY(): The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country". The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

Syntax: SELECT *column_name(s)* FROM *table_name* WHERE *condition*
GROUP BY *column_name(s)* ORDER BY *column_name(s);*

Find count of all rows group by each Company?

```
SQL> select compname,count(*) from company group by compname;
COMPNAME             COUNT(*)
-------------------- ----------
Wipro                     2
IBM                       1
Dell                      2
```

Find sum of amount of each Company?

```
SQL> select compname,sum(amount) from company group by compname;
COMPNAME             SUM(AMOUNT)
-------------------- -----------
Wipro                   9000
IBM                     9000
Dell                   16000
```

Find maximum amount of each Company?

```
SQL> select compname,max(amount) from company group by compname;
COMPNAME             MAX(AMOUNT)
-------------------- -----------
Wipro                    5000
IBM                      9000
Dell                    10000
```

Find minimum amount of each Company?

```
SQL> select compname,min(amount) from company group by compname;
COMPNAME             MIN(AMOUNT)
-------------------- -----------
Wipro                       4000
IBM                         9000
Dell                        6000
```

ORDER BY(): The ORDER BY keyword is used to sort the result-set in ascending or descending order. The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.
Syntax: SELECT *column1*, *column2*, ... FROM *table_name* ORDER BY *column1*, *column2*, ... ASC|DESC;

Find count of all rows group by each Company order by Company Name?

```
SQL> select compname,count(*) from company group by compname order by compname;
COMPNAME             COUNT(*)
-------------------- ----------
Dell                          2
IBM                           1
Wipro                         2
```

HAVING(): The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
Syntax: SELECT *column_name(s)* FROM *table_name* WHERE *condition*
GROUP BY *column_name(s)* HAVING *condition* ORDER BY *column_name(s);*

Find sum of amount of each Company and having sum of amount greater than 10000?

```
SQL> select compname,sum(amount) from company group by compname having
sum(amount)>10000;
COMPNAME             SUM(AMOUNT)
-------------------- -----------
Dell                       16000
```

Find the count of all rows grouped by each Company name and having count greater than 1?

```
SQL> select compname,count(*) from company group by compname having count(*)>1;
COMPNAME             COUNT(*)
-------------------- ----------
Wipro                         2
Dell                          2
```

Find the count of each company and display in sorted order according to company?

```
SQL> select compname,count(*) from company group by compname order by
compnameasc;
COMPNAME             COUNT(*)
-------------------- ----------
Dell                          2
IBM                           1
Wipro                         2
```

Find the name and age of sailors who have reserved a red boat and list the order of age?

```
SQL> select s.sname,s.age from sailors s,boatsb,reserves r where s.sid=r.sid and b.bid=r.bid
and b.color='Red' order by s.age;
SNAME                          AGE
------------------------------ ----------
Horatio                         35
Dustin                          45
Dustin                          45
Lubber                         55.5
Lubber                         55.5
```

Find the average age of sailors for each rating level?

```
SQL> select s.rating,avg(s.age) from sailors s group by s.rating;
   RATING AVG(S.AGE)
---------- ----------
         1         33
         8       40.5
         7         40
         3      44.25
        10       25.5
         9         35
6 rows selected.
```

Find the minimum age of sailors for each rating level?

```
SQL> select rating,min(age) from sailors group by rating;
   RATING   MIN(AGE)
---------- ----------
         1         33
         8       25.5
         7         35
         3         25
        10         16
         9         35
6 rows selected.
```

Find the maximum age of sailors for each rating level?

```
SQL> select rating,max(age) from sailors group by rating;
   RATING   MAX(AGE)
---------- ----------
         1         33
         8       55.5
```

7	45
3	63.5
10	35
9	35

6 rows selected.

Find the average age of sailors for each rating level that has atleast two subsailors?
SQL> select rating,avg(age) from sailors group by rating having count(*)>1;

RATING	AVG(AGE)
8	40.5
7	40
3	44.25
10	25.5

Find the name and age of the oldest sailor?
SQL> select s.sname,s.age from sailors s where(select max(s1.age) from sailors s1)=s.age;

SNAME	AGE
Bob	63.5

4.2) Queries on Controlling Data: Commit, Rollback, and Save point.

COMMIT: COMMIT in SQL is a transaction control language that is used to permanently save the changes done in the transaction in tables/databases. The database cannot regain its previous state after its execution of commit.
Syntax: commit;

ROLLBACK: ROLLBACK in SQL is a transactional control language that is used to undo the transactions that have not been saved in the database. The command is only been used to undo changes since the last COMMIT.
Syntax: rollback; (or) rollback to savepoint_name;

SAVEPOINT: creates points within the groups of transactions in which to ROLLBACK. A SAVEPOINT is a point in a transaction in which you can roll the transaction back to a certain point without rolling back the entire transaction.
Syntax: SAVEPOINT SAVEPOINT_NAME;

```
SQL> create table student
  2  (
  3  rno integer,
  4  name varchar2(20)
  5  );
Table created.

SQL> insert into student values(501,'Tarun');
1 row created.

SQL> insert into student values(502,'Udaya');
1 row created.

SQL> savepoint a;
Savepoint created.

SQL> insert into student values(503,'Raghu');
1 row created.

SQL> savepoint b;
Savepoint created.

SQL> insert into student values(504,'Geetha');
1 row created.

SQL> select * from student;
      RNO NAME
---------- --------------------
       501 Tarun
       502 Udaya
       503 Raghu
```

504 Geetha

```
SQL> rollback to b;
Rollback complete.

SQL> select * from student;
      RNO NAME
---------- --------------------
       501 Tarun
       502 Udaya
       503 Raghu

SQL> rollback to a;
Rollback complete.

SQL> select * from student;
      RNO NAME
---------- --------------------
       501 Tarun
       502 Udaya

SQL> commit;
Commit complete.

SQL> rollback;
Rollback complete.

SQL> select * from student;
      RNO NAME
---------- --------------------
       501 Tarun
       502 Udaya
```

Experiment-5:

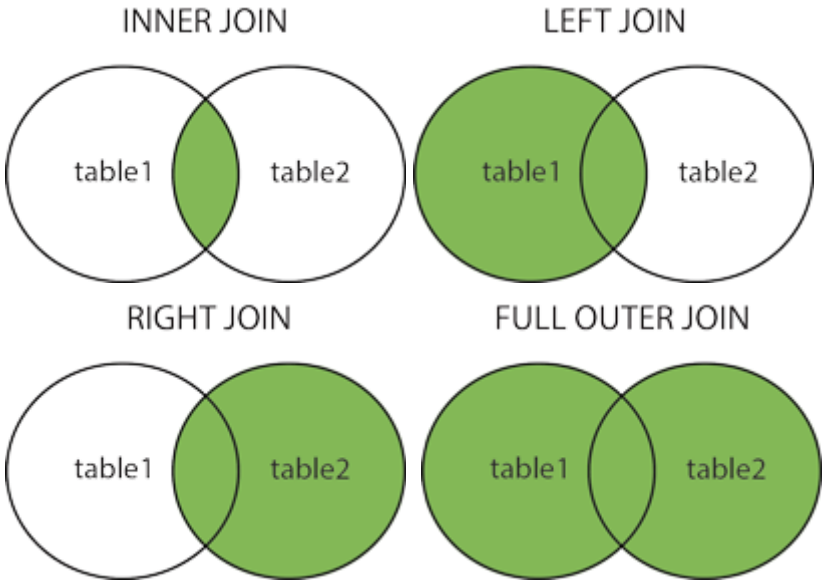
5) Queries on Joins and Correlated Sub-queries.

JOINS

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Here are the different types of the JOINs in SQL:

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table




```
SQL> create table th1
2  (
3  rno integer,
4  name varchar2(30),
5  marks integer
6  );
Table created.

SQL> create table th2
2  (
3  rno integer,
4  fee integer
5  );
Table created.

SQL> select * from th1;
      RNO NAME                           MARKS
---------- ------------------------------ ----------
       501 abhi                           50
       502 ravi                           40
```

```
     503 suma                       30
     504 raju                      35
     505 ramu                       45

SQL> select * from th2;
     RNO       FEE
---------- ----------
     501       3000
     502       2000
     503       1500
     504       4000

SQL> commit;
Commit complete.
```

INNER JOINING TWO TABLES

INNER JOIN Syntax:

SELECT *column_name(s)*
FROM *table1*
INNER JOIN *table2*
ON *table1.column_name = table2.column_name*;

```
SQL> select * from th1 inner join th2 on th1.rno=th2.rno;
     RNO NAME                          MARKS       RNO       FEE
---------- ------------------------------ ---------- ---------- ----------
     501 abhi                           50       501      3000
     502 ravi                          40       502      2000
     503 suma                           30       503      1500
     504 raju                          35       504      4000

SQL> select * from th1 join th2 on th1.rno=th2.rno;
     RNO NAME                          MARKS       RNO       FEE
---------- ------------------------------ ---------- ---------- ----------
     501 abhi                           50       501      3000
     502 ravi                          40       502      2000
     503 suma                           30       503      1500
     504 raju                          35       504      4000
```

LEFT OUTER JOIN

LEFT JOIN Syntax:

SELECT *column_name(s)*
FROM *table1*
LEFT JOIN *table2*
ON *table1.column_name = table2.column_name*;

```
SQL> select * from th1 left outer join th2 on th1.rno=th2.rno;
     RNO NAME                          MARKS       RNO       FEE
---------- ------------------------------ ---------- ---------- ----------
     501 abhi                           50       501      3000
     502 ravi                          40       502      2000
     503 suma                           30       503      1500
     504 raju                          35       504      4000
     505 ramu                           45
```

RIGHT OUTER JOIN

RIGHT JOIN Syntax:

SELECT *column_name(s)*
FROM *table1*
RIGHT JOIN *table2*
ON *table1.column_name = table2.column_name*;

SQL> select * from th2 right outer join th1 on th2.rno=th1.rno;

RNO	FEE	RNO	NAME	MARKS
501	3000	501	abhi	50
502	2000	502	ravi	40
503	1500	503	suma	30
504	4000	504	raju	35
		505	ramu	45

NATURAL JOIN

The SQL NATURAL JOIN is a type of EQUI JOIN and is structured in such a way that, columns with the same name of associated tables will appear once only.

NATURAL JOIN Syntax:

SELECT *
FROM table1
NATURAL JOIN table2;

SQL> select * from th1 natural join th2;

RNO	NAME	MARKS	FEE
501	abhi	50	3000
502	ravi	40	2000
503	suma	30	1500
504	raju	35	4000

CROSS JOIN

The CROSS JOIN keyword returns all records from both tables (table1 and table2).

CROSS JOIN Syntax:

SELECT *column_name(s)*
FROM *table1*
CROSS JOIN *table2*;

SQL> select * from th1 cross join th2;

RNO	NAME	MARKS	RNO	FEE
501	abhi	50	501	3000
502	ravi	40	501	3000
503	suma	30	501	3000
504	raju	35	501	3000
505	ramu	45	501	3000
501	abhi	50	502	2000
502	ravi	40	502	2000
503	suma	30	502	2000
504	raju	35	502	2000
505	ramu	45	502	2000

```
     501 abhi      50       503      1500
     RNO NAME                      MARKS       RNO       FEE
---------- ------------------------------ ---------- ---------- ----------
     502 ravi                        40       503      1500
     503 suma                         30       503      1500
     504 raju                        35       503      1500
     505 ramu                         45       503      1500
     501 abhi                        50       504      4000
     502 ravi                        40       504      4000
     503 suma                         30       504      4000
     504 raju                        35       504      4000
     505 ramu      45       504      4000
20 rows selected.

SQL> select * from th1,th2;
     RNO NAME                      MARKS       RNO       FEE
---------- ------------------------------ ---------- ---------- ----------
     501 abhi                        50       501      3000
     502 ravi                        40       501      3000
     503 suma                         30       501      3000
     504 raju                        35       501      3000
     505 ramu                         45       501      3000
     501 abhi                        50       502      2000
     502 ravi                        40       502      2000
     503 suma                         30       502      2000
     504 raju                        35       502      2000
     505 ramu                         45       502      2000
     501 abhi      50       503      1500
     RNO NAME                      MARKS       RNO       FEE
---------- ------------------------------ ---------- ---------- ----------
     502 ravi                        40       503      1500
     503 suma                         30       503      1500
     504 raju                        35       503      1500
     505 ramu                         45       503      1500
     501 abhi                        50       504      4000
     502 ravi                        40       504      4000
     503 suma                         30       504      4000
     504 raju                        35       504      4000
     505 ramu      45       504      4000
20 rows selected.
```

IN and NOT IN

It tests whether a value is in a given set of elements or not.

Syntax: select column_list from table_list where col_name IN[NOT IN] (selectcol_name from table_name where condition);

ALL and ANY

Used to compare a value to a list. It is preceded by comparision operator and followed by a list.

Syntax: select column_list from table_list where col_name comparision_operator ALL/ANY (sub_query);

Nested Queries

A *nested* query is a complete query embedded within another operation. A nested query can have all the elements used in a regular query, and any valid query can be embedded within another operation to become a nested query. For instance, a nested query can be embedded within INSERT and DELETE operations. Depending on the operation, a nested query should be embedded by enclosing the statement within the correct number of parentheses to follow a particular order of operations. A nested query is also useful in scenarios where you want to execute multiple commands in one query statement, rather than writing multiple ones to return your desired result(s).

Find the name and age of the youngest sailor?

```
SQL> select s.sname,s.age from sailors s where s.age<=all(select s.age from sailors s);
SNAME                          AGE
------------------------------ ----------
Zobra                          16
```

Find the name of the sailors who has the highest rating?

```
SQL> select s.sname from sailors s where s.rating>=all(select s1.rating from sailors s1);
SNAME
------------------------------
Zobra
Rusty
```

Find the names of sailors whose rating is better than Horatio?

```
SQL> select s.sname from sailors s where s.rating>all(select s1.rating from sailors s1 where
s1.sname='Horatio');
SNAME
------------------------------
Lubber
Andy
Zobra
Ravi
Rusty
```

Multiple Nested Queries

Multiple-row subqueries are nested queries that can return more than one row of results to the parent query. Multiple-row subqueries are used most commonly in WHERE and HAVING clauses. Since it returns multiple rows, it must be handled by set comparison operators (IN, ALL, ANY). While, IN operator holds the same meaning as discussed in earlier chapter, ANY operator compares a specified value to each value returned by the sub query while ALL compares a value to every value returned by a sub query.

Find the names of the sailors who have reserved a red boat using IN operator?

```
SQL> select s.sname from sailors s where s.sid in(select r.sid from reserves r where r.bid
in(select b.bid from boats b where b.color='Red'));
SNAME
------------------------------
Dustin
Lubber
Horatio
```

Find the names of sailors who doesn't have reserved a red boat using NOT IN operator?
SQL> select s.sname from sailors s where s.sid not in(select r.sid from reserves r where r.bid in(selectb.bid from boats b where b.color='Red'));
SNAME

Brutus
Andy
Zobra
Ravi
Art
Bob
Rusty
7 rows selected.

Find the names of sailors who have reserved a boat number 103?
SQL> select s.sname from sailors s where s.sid in(select r.sid from reserves r where r.bid=103);
SNAME

Dustin
Lubber
Ravi

Find the names of sailors who have not reserved boat number 103?
SQL> select s.sname from sailors s where s.sid not in(select r.sid from reserves r where r.bid=103);
SNAME

Brutus
Andy
Horatio
Zobra
Art
Bob
Rusty
7 rows selected.

Correlated Queries

SQL Correlated Subqueries are used to select data from a table referenced in the outer query. The subquery is known as a correlated because the subquery is related to the outer query. In this type of queries, a table alias (also called a correlation name) must be used to specify which table reference is to be used.

Exists Operator:

The EXISTS operator tests for existence of rows in the results set of the subquery. If a subquery row value is found, the condition is flagged TRUE and the search doesn't continue in the inner query. And if it is not found then the condition is flagged FALSE and the search continues in the inner query.

Not Exists Operator:

The SQL NOT EXISTS Operator will act quite opposite to EXISTS Operator. It is used to restrict the number of rows returned by the SELECT Statement.

Find the names of the sailors who have reserved boat number 103 using EXISTS?

```
SQL> select s.sname from sailors s where exists(select * from reserves r where r.sid=s.sid
and r.bid=103);
SNAME
------------------------------
Dustin
Lubber
Ravi
```

Find the names of the sailors who have not reserved boat number 103 using NOT EXISTS?

```
SQL> select s.sname from sailors s where not exists(select * from reserves r where r.sid=s.sid
and r.bid=103);
SNAME
------------------------------
Brutus
Andy
Rusty
Horatio
Zobra
Art
Bob
```

Find distinct names of sailors who have reserved atleast one boat?

```
SQL> select distinct(s.sname) from sailors s,reserves r where s.sid=r.sid;
SNAME
------------------------------
Lubber
Ravi
Dustin
Horatio
```

Find the names of the sailors who have reserved both red and green boats without using INTERSECTION?

```
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';
SNAME
------------------------------
Dustin
Dustin
Lubber
Lubber
```

Find the names of the sailors who have reserved a red boat or a green boat without using UNION?

```
SQL> select s.sname from sailors s,reservesr,boats b where s.sid=r.sid and r.bid=b.bid
and(b.color='Red' or b.color='Green');
SNAME
------------------------------
Dustin
```

Dustin
Dustin
Lubber
Lubber
Lubber
Horatio
Ravi

Find the sailor name, boat id and reservation date for each reservation?
SQL> select s.sname,r.bid,r.day from sailors s,reserves r where s.sid=r.sid;

SNAME	BID	DAY
Dustin	101	10-OCT-98
Dustin	102	10-OCT-98
Dustin	103	10-AUG-98
Dustin	104	10-JUL-98
Lubber	102	11-OCT-98
Lubber	103	11-JUN-98
Lubber	104	11-DEC-98
Horatio	101	09-MAY-98
Horatio	102	09-AUG-98
Ravi	103	09-AUG-98

Find the names of sailors who are elder than oldest sailor with a rating of 10?
SQL> select s.sname from sailors s where s.age>(select max(s1.age) from sailors s1 where s1.rating=10);

SNAME
Dustin
Lubber
Bob

Find the average age of sailors for each rating level that has atleast two sailors?
SQL> select s.rating,avg(s.age) as avgage from sailors s group by s.rating having count(*)>1;

RATING	AVGAGE
8	40.5
7	40
3	44.25
10	25.5

Find age of youngest sailor with age greater than 18, for each rating with atleast two 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;

RATING	MINAGE
8	25.5
7	35
3	25

Experiment-6: Queries on Working with Index, Sequence, Synonyms.

INDEX

CREATE INDEX:

The CREATE INDEX statement is used to create indexes in tables. Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries.

Syntax:

CREATE INDEX *index_name*
ON *table_name* (*column1*, *column2*, ...);

SQL> create index ind on student(UPPER(name));
Index created.

ALTER INDEX:

The ALTER INDEX statement is used to alter the definition of an index.

Syntax: alter index old_index_name rename to new_index_name;

SQL> alter index ind rename to inde;
Index altered.

DROP INDEX:

The DROP INDEX statement is used to delete an index in a table.

SYNTAX:

DROP INDEX *index_name*;

SQL> drop index inde;
Index dropped.

SEQUENCE

Sequence is a set of integers 1, 2, 3, … that are generated and supported by some database systems to produce unique values on demand.

- A sequence is a user defined schema bound object that generates a sequence of numeric values.
- Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.
- The sequence of numeric values is generated in an ascending or descending order at defined intervals and can be configured to restart when exceeds max_value.

Syntax:

CREATE SEQUENCE sequence_name
START WITH initial_value
INCREMENT BY increment_value
MINVALUE minimum value
MAXVALUE maximum value
CYCLE|NOCYCLE;

sequence_name: Name of the sequence.
initial_value: starting value from where the sequence starts.
Initial_value should be greater than or equal to minimum value and less than equal to maximum value.

increment_value: Value by which sequence will increment itself.
Increment_value can be positive or negative.
minimum_value: Minimum value of the sequence.
maximum_value: Maximum value of the sequence.
cycle: When sequence reaches its set_limit it starts from beginning.
nocycle: An exception will be thrown if sequence exceeds its max_value.

```
SQL> create sequence s1 start with 1 increment by 1;
Sequence created.

SQL> insert into student(sid,sname,age)values(s1.nextval,'Jungkook',24);
1 row created.

SQL> insert into student(sid,sname,age)values(s1.nextval,'Felix',19);
1 row created.

SQL> select * from student;
      SID SNAME                          AGE
---------- ------------------------------ ----------
         1 Jungkook                       24
         2 Felix                          19

SQL> alter sequence s1 maxvalue 200;
Sequence altered.

SQL> select * from user_sequences;
SEQUENCE_NAME                 MIN_VALUE  MAX_VALUE INCREMENT_BY C O
CACHE_SIZE
------------------------------ ---------- ---------- ------------ - - ----------
LAST_NUMBER
-----------
S1                                     1       200            1 N N        20
         3

SQL> drop sequence s1;
Sequence dropped.
```

SYNONYMS

A SYNONYM provides another name for database object, referred to as original object, that may exist on a local or another server. A synonym belongs to schema, name of synonym should be unique. A synonym cannot be original object for an additional synonym and synonym cannot refer to user-defined function.
The query below results in an entry for each synonym in database. This query provides details about synonym metadata such as the name of synonym and name of the base object.

Syntax:

```
CREATE SYNONYM synonymname
FOR servername.databasename.schemaname.objectname;
```

An alias or alternative names can be given to any of the database objects like a table, view, stored procedure, user-defined function, and sequence with the help of SQL Server Synonym.

Whenever we create a SQL Server Synonym in a database, the synonym is referenced to a particular database object and that database object is called base object. The location of the base object to which the synonym is referenced can be either in the same database or in some other database in the same server or even on some other instance running on another server.

We can use SQL Server Synonym in various scenarios and take advantage out of them. Some of the scenarios are:

- As there are 100's or sometimes 1000's of references in code to a particular object, so in that case, we can assign a synonym to that object.
- An additional layer of abstraction can be added to the actual base object to which the synonym is assigned.
- The database queries can easily refer to the objects that appear to reside in the current database even though in reality they are present in some other database.
- Backward compatibility can also be provided to the old legacy base object that is needed in the newer version of the database.
- An additional security layer for the protection of the base object can be provided with the help of SQL Server Synonym.
- There are scenarios when we want to move to another database, with the help of SQL Server Synonym we can refer to the base object without thinking about the migration.
- The lengthy and confusing object names of a database can be simplified with the help of SQL Server Synonym.
- Various issues with the cross-database and server dependencies in downstream environments such as development, test, and quality assurance as part of a continuous integration build process can be easily eliminated with the use of the SQL Server Synonym.

The syntax for creating a Synonym in SQL Server Database is:

CREATE SYNONYM [name_of_schema.] name_of_synonym
FOR name_of_base_object;

In the above-written syntax:
Name_of_schema: The name_of_schema is the name of the schema where the synon
ym is going to be created.
Name_of_synonym: The name_of_synonym represents the name of the synonym.
Name_of_base_object: The name_of_base_object represents the name of the base object to which the synonym is going to be assigned.

Experiment-7: Queries to Build Views.

VIEWS AND CREATION OF VIEWS

In SQL, a view is a virtual table based on the result-set of an SQL statement.
A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database. You can add SQL statements and functions to a view and present the data as if the data were coming from one single table.
A view is created with the CREATE VIEW statement.

CREATE VIEW Syntax:

CREATE VIEW *view_name* AS SELECT *column1, column2, ...* FROM *table_name* WHERE *condition*;

```
SQL> create view sail_view as(select sid,sname from sailors);
View created.

SQL> desc sail_view;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 SID                                       NOT NULL NUMBER(38)
 SNAME                                              VARCHAR2(30)

SQL> select * from sail_view;
       SID SNAME
---------- ------------------------------
        22 Dustin
        29 Brutus
        31 Lubber
        32 Andy
        64 Horatio
        71 Zobra
        74 Ravi
        85 Art
        95 Bob
        58 Rusty
10 rows selected.

SQL> insert into sail_view values(100,'Uday');
1 row created.

SQL> select * from sail_view;
       SID SNAME
---------- ------------------------------
       100 Uday
        22 Dustin
        29 Brutus
        31 Lubber
        32 Andy
        64 Horatio
        71 Zobra
        74 Ravi
        85 Art
```

95 Bob
58 Rusty
11 rows selected.

SQL> delete from sail_view where sid=100;
1 row deleted.

SQL> select * from sail_view;

SID	SNAME
22	Dustin
29	Brutus
31	Lubber
32	Andy
64	Horatio
71	Zobra
74	Ravi
85	Art
95	Bob
58	Rusty

10 rows selected.

CREATE OR REPLACE VIEW

A view can be updated with the CREATE OR REPLACE VIEW statement.

CREATE OR REPLACE VIEW Syntax:

CREATE OR REPLACE VIEW *view_name* AS SELECT *column(s)* FROM *table_name* WHERE *condition*;

SQL> create or replace view sail_view as(select sid,rating from sailors);
View created.

SQL> select * from sail_view;

SID	RATING
22	7
29	1
31	8
32	8
64	7
71	10
74	9
85	3
95	3
58	10

10 rows selected.

READ ONLY VIEW Creation

We can create a view with read-only option to restrict access to the view.
Syntax to create a view with Read-Only Access

Syntax:

CREATE or REPLACE FORCE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition WITH read-only;

The above syntax will create view for **read-only** purpose, we cannot Update or Insert data into read-only view. It will throw an **error**.

```
SQL> create view sail_stu as select rno,name from th1 WITH READ ONLY;
View created.

SQL> insert into sail_stu values(45,'Ooha');
insert into sail_stu values(45,'Ooha')
*
ERROR at line 1:
ORA-01733: virtual column not allowed here

SQL> update sail_stu
  2  set name='Teja'
  3  where rno=501;
set name='Teja'
    *
ERROR at line 2:
ORA-01733: virtual column not allowed here
```

FORCE VIEW Creation

FORCE keyword is used while creating a view, forcefully. This keyword is used to create a View even if the table does not exist. After creating a force View if we create the base table and enter values in it, the view will be automatically updated.
Syntax for forced View is:
CREATE or REPLACE FORCE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition;

```
SQL> create force view vi_stu as select * from dummy;
Warning: View created with compilation errors.
```

CHECK OPTION VIEW Creation

WITH CHECK OPTION is an optional clause on the CREATE VIEW statement. It specifies the level of checking when data is inserted or updated through a view.
If WITH CHECK OPTION is specified, every row that is inserted or updated through the view must conform to the definition of the view. The option cannot be specified if the view is read-only. The definition of the view must not include a subquery.

```
SQL> create view sail_stu1 as select * from th1 where marks<=100 WITH CHECK
OPTION;
View created.

SQL> insert into sail_stu1 values(12,'Padma',191);
insert into sail_stu1 values(12,'Padma',191)
            *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation
```

Experiment-8: Write a PL/SQL Code using Basic Variables and Usage of Assignment Operation.

PL/SQL is a procedural language

Three sections

1. Declaration section: declare the variables, exceptions, cursor
2. Execution section: the execution section always starts with the BEGIN keyword and ends with the END keyword
3. Exception-Handling section: it starts with the EXCEPTION keyword.

Syntax:

```
DECLARE
  Declaration statements;
BEGIN
  Execution statements;
 EXCEPTION
    Exception handling statements;
END;
/
```

Each variable in the PL/SQL has a specific data type which defines the size and layout of the variable's memory.

variable_name datatype:= initial_value

Here, variable_name is a valid identifier in PL/SQL and datatype must be valid PL/SQL data type.

Assignment operator is to set a variable equal to the value or expression on the other side of the operator.

```
SQL> set serveroutput on
```

Write a PL/SQL program to print Hello World?

```
SQL> begin
  2  dbms_output.put_line('Hello World');
  3  end;
  4  /
Hello World
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to add two numbers?

```
SQL> declare
  2  a integer;
  3  b integer;
  4  c integer;
  5  begin
  6  a:=2;
  7  b:=3;
  8  c:=a+b;
  9  dbms_output.put_line('Value of a is:'||a);
 10  dbms_output.put_line('Value of b is:'||b);
 11  dbms_output.put_line('Value of c is:'||c);
 12  end;
 13  /
Value of a is:2
```

Value of b is:3
Value of c is:5
PL/SQL procedure successfully completed.

Write a PL/SQL program to add two numbers of input from the user?

```
SQL> declare
  2  a integer:=&a;
  3  b integer:=&b;
  4  c integer;
  5  begin
  6  c:=a+b;
  7  dbms_output.put_line('Value of a is:'||a);
  8  dbms_output.put_line('Value of b is:'||b);
  9  dbms_output.put_line('Value of c is:'||c);
 10  end;
 11  /
Enter value for a: 34
old   2: a integer:=&a;
new   2: a integer:=34;
Enter value for b: 35
old   3: b integer:=&b;
new   3: b integer:=35;
Value of a is:34
Value of b is:35
Value of c is:69
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to display student details?

```
SQL> declare
  2  name varchar2(20);
  3  rno integer;
  4  branch varchar2(5);
  5  gender char(1);
  6  begin
  7  name:='Harsha';
  8  rno:=2317;
  9  branch:='CSE';
 10  gender:='M';
 11  dbms_output.put_line(name||' '||rno||' '||branch||' '||gender);
 12  end;
 13  /
Harsha 2317 CSE M
PL/SQL procedure successfully completed.
```

Experiment-9: Write a PL/SQL Code to Bind and Substitute variables in PL/SQL.

BIND VARIABLES

Bind variables are variables you create in SQL*Plus and then reference in PL/SQL. If you create a bind variable in SQL*Plus, you can use the variable as you would a declared variable in your PL/SQL subprogram and then access the variable from SQL*Plus. You can use bind variables for such things as storing return codes or debugging your PL/SQL subprograms. Because bind variables are recognized by SQL*Plus, you can display their values in SQL*Plus or reference them in other PL/SQL subprograms that you run in SQL*Plus.

Creating Bind Variables:

You create bind variables in SQL*Plus with the VARIABLE command. For example,

Syntax: VARIABLE ret_val NUMBER

This command creates a bind variable named ret_val with a datatype of NUMBER.

Referencing Bind Variables:

You reference bind variables in PL/SQL by typing a colon (:) followed immediately by the name of the variable. For example

Syntax: :ret_val := 1;

To change this bind variable in SQL*Plus, you must enter a PL/SQL block.

Displaying Bind Variables:

To display the value of a bind variable in SQL*Plus, you use the SQL*Plus PRINT command. For example

Syntax: print ret_val;

```
SQL> variable a number

SQL> begin
  2  :a:=1;
  3  end;
  4  /
PL/SQL procedure successfully completed.

SQL> print a;
         A
----------
         1

SQL> exec:a:=2;
PL/SQL procedure successfully completed.

SQL> print a;
         A
----------
         2
```

SUBSTITUTE VARIABLES

PL/SQL blocks can accept user input while execution. This provision is available with substitution variables. They are not allocated any memory, so they can't be used for returning values as output. If you want to return values from a named PL/SQL block then you can use OUT parameters in procedures or return value from functions.
Whenever PL/SQL processor encounters a substitution variable in a block, it halts for user input. All the substitution variables in the block are replaced by the inputted values before sending it to the database.
Substitution variables are used by prefixing a variable name with the ampersand(&) or double ampersand (&&).
Ampersand(&) is used when you want to give a new value each time a substitution variable is encountered in the code. The user has to input value even if the same substitution variable is used at multiple places in a PL/SQL program.
Double ampersand (&&) is used when you want the same value to be used for all occurrences of a variable in PL/SQL program. With &&, PL/SQL processor asks only once for each substitution variable. Wherever the substitution variable is encountered in the code it is replaced with previously assigned value.

DEFINE Statement:
The DEFINE statement will be used to define the value for substitution variable
Syntax:
define substitution_variable:=value;

UNDEFINE Statement:
The UNDEFINE statement will be used to UNDIFINE the variable
Syntax:
undefine substitution_variable;

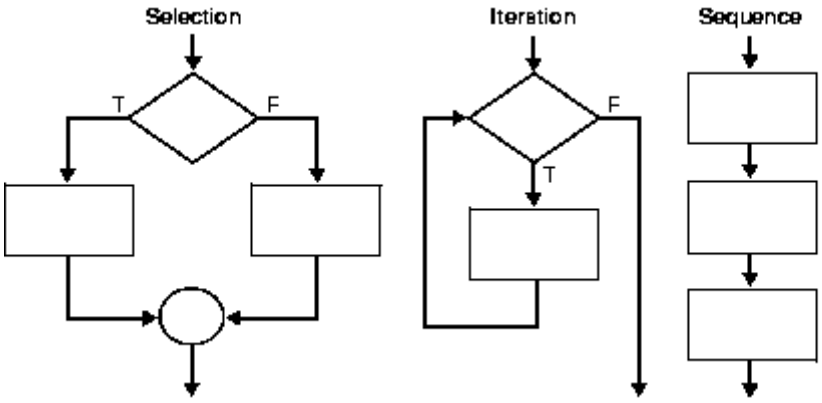
```
SQL> define name='Ravi';
SQL> select '&&name' from dual;
old   1: select '&&name' from dual
new   1: select 'Ravi' from dual
'RAV
----
Ravi

SQL> undefine name;
SQL> select '&&name' from dual;
Enter value for name: Ravi
old   1: select '&&name' from dual
new   1: select 'Ravi' from dual
'RAV
----
Ravi
```

Experiment-10: Write a PL/SQL block using SQL and Control Structures.

SQL> set serveroutput on;

If-Else

According to the *structure theorem*, any computer program can be written using the basic control structures as shown. They can be combined in any way necessary to deal with a given problem.



The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A *condition* is any variable or expression that returns a Boolean value (TRUE or FALSE). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Conditional Control: IF and CASE Statements:

Often, it is necessary to take alternative actions depending on circumstances.
The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions.

IF-THEN Statement:

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF), as follows:

Syntax:

```
IF condition THEN
  sequence_of_statements
END IF;
```

The sequence of statements is executed only if the condition is true. If the condition is false or null, the IF statement does nothing. In either case, control passes to the next statement.

IF-THEN-ELSE Statement:

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

Syntax:

```
IF condition THEN
  sequence_of_statements1
ELSE
  sequence_of_statements2
END IF;
```

The sequence of statements in the ELSE clause is executed only if the condition is false or null. Thus, the ELSE clause ensures that a sequence of statements is executed.

Write a PL/SQL program to check a number is even or odd?

```
SQL> declare
  2  n integer:=&n;
  3  begin
  4  if mod(n,2)=0 then
  5  dbms_output.put_line('Even Number '||n);
  6  else
  7  dbms_output.put_line('Odd Number '||n);
  8  end if;
  9  end;
 10  /
Enter value for n: 4
old   2: n integer:=&n;
new   2: n integer:=4;
Even Number 4
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print the biggest out of two numbers?

```
SQL> declare
  2  a integer:=&a;
  3  b integer:=&b;
  4  begin
  5  if(a>b) then
  6  dbms_output.put_line(a||' is bigger');
  7  else
  8  dbms_output.put_line(b||' is bigger');
  9  end if;
 10  end;
 11  /
Enter value for a: 5
old   2: a integer:=&a;
new   2: a integer:=5;
Enter value for b: 4
old   3: b integer:=&b;
new   3: b integer:=4;
5 is bigger
PL/SQL procedure successfully completed.
```

If-Else if

IF-THEN-ELSIF Statement:

Sometimes you want to select an action from several mutually exclusive alternatives. The third form of IF statement uses the keyword ELSIF (not ELSEIF) to introduce additional conditions, as follows:

Syntax:

```
IF condition1 THEN
   sequence_of_statements1
ELSIF condition2 THEN
```

```
  sequence_of_statements2
ELSE
  sequence_of_statements3
END IF;
```

If the first condition is false or null, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is true, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or null, the sequence in the ELSE clause is executed.

Write a PL/SQL program to take marks as input and print status?

```
SQL> declare
  2  marks integer:=&marks;
  3  begin
  4  if(marks>=75) then
  5  dbms_output.put_line('Distinction');
  6  elsif(marks>=60 and marks<75) then
  7  dbms_output.put_line('First Class');
  8  elsif(marks>=50 and marks<60) then
  9  dbms_output.put_line('Second Class');
 10  else
 11  dbms_output.put_line('Fail');
 12  end if;
 13  end;
 14  /
Enter value for marks: 99
old   2: marks integer:=&marks;
new   2: marks integer:=99;
Distinction
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to find the biggest out of three numbers?

```
SQL> declare
  2  a integer:=&a;
  3  b integer:=&b;
  4  c integer:=&c;
  5  begin
  6  if(a>b and a>c) then
  7  dbms_output.put_line(a||' is bigger');
  8  elsif(b>a and b>c) then
  9  dbms_output.put_line(b||' is bigger');
 10  else
 11  dbms_output.put_line(c||' is bigger');
 12  end if;
 13  end;
 14  /
Enter value for a: 1
old   2: a integer:=&a;
new   2: a integer:=1;
```

Enter value for b: 2
old 3: b integer:=&b;
new 3: b integer:=2;
Enter value for c: 3
old 4: c integer:=&c;
new 4: c integer:=3;
3 is bigger
PL/SQL procedure successfully completed.

Switch Cases

CASE Statement:

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions.

The CASE statement is more readable and more efficient. So, when possible, rewrite lengthy IF-THEN-ELSIF statements as CASE statements.

The CASE statement begins with the keyword CASE. The selector expression can be arbitrarily complex. The selector is followed by one or more WHEN clauses, which are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a WHEN-clause expression, that WHEN clause is executed.

The ELSE clause works similarly to the ELSE clause in an IF statement. In the last example, if the grade is not one of the choices covered by a WHEN clause, the ELSE clause is selected, and the phrase 'No such grade' is output. The ELSE clause is optional.

If the CASE statement selects the implicit ELSE clause, PL/SQL raises the predefined exception CASE_NOT_FOUND. So, there is always a default action, even when you omit the ELSE clause.

The keywords END CASE terminate the CASE statement. These two keywords must be separated by a space. The CASE statement has the following form:

Syntax:

```
[<<label_name>>]
CASE selector
   WHEN expression1 THEN sequence_of_statements1;
   WHEN expression2 THEN sequence_of_statements2;
   ...
   WHEN expressionN THEN sequence_of_statementsN;
  [ELSE sequence_of_statementsN+1;]
END CASE [label_name];
```

Write a PL/SQL program to take marks as input and print the status using cases?

```
SQL> declare
  2  grade char(1):='&grade';
  3  begin
  4  case grade
  5  when 'A' then dbms_output.put_line('Excellent');
  6  when 'B' then dbms_output.put_line('Good');
  7  when 'C' then dbms_output.put_line('Average');
  8  when 'D' then dbms_output.put_line('Bad');
  9  else
 10  dbms_output.put_line('Wrong Input');
```

```
 11  end case;
 12  end;
 13  /
Enter value for grade: A
old   2: grade char(1):='&grade';
new   2: grade char(1):='A';
Excellent
PL/SQL procedure successfully completed.
```

Nested-If

Write a PL/SQL program to print the biggest of 3 numbers?

```
SQL> declare
  2  a int:=&a;
  3  b int:=&b;
  4  c int:=&c;
  5  begin
  6  if(a>b) then
  7  if(a>c) then dbms_output.put_line(a||' is bigger');
  8  else dbms_output.put_line(c||' is bigger');
  9  end if;
 10  else
 11  if(b>c) then dbms_output.put_line(b||' is bigger');
 12  else dbms_output.put_line(c||' is bigger');
 13  end if;
 14  end if;
 15  end;
 16  /
Enter value for a: 4
old   2: a int:=&a;
new   2: a int:=4;
Enter value for b: 5
old   3: b int:=&b;
new   3: b int:=5;
Enter value for c: 6
old   4: c int:=&c;
new   4: c int:=6;
6 is bigger
PL/SQL procedure successfully completed.
```

Loops

Simple Loop

Iterative Control: LOOP and EXIT Statements:

LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

LOOP:

The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

Syntax:

```
LOOP
  sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop.

Write a PL/SQL program to print a sequence of n numbers using simple loop?

```
SQL> declare
  2  a int:=1;
  3  n int:=&n;
  4  begin
  5  loop
  6  dbms_output.put_line(a);
  7  a:=a+1;
  8  exit when a>n;
  9  end loop;
 10  end;
 11  /
Enter value for n: 5
old   3: n int:=&n;
new   3: n int:=5;
1
2
3
4
5
PL/SQL procedure successfully completed.
```

While Loop

WHILE-LOOP:

The WHILE-LOOP statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

Syntax:

```
WHILE condition LOOP
  sequence_of_statements
END LOOP;
```

Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement. The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times.

Write a PL/SQL program to print a sequence of n numbers using while loop?

```
SQL> declare
  2  a int:=1;
  3  n int:=&n;
  4  begin
  5  while(a<=n)loop
```

```
  6  dbms_output.put_line(a);
  7  a:=a+1;
  8  end loop;
  9  end;
 10  /
Enter value for n: 5
old   3: n int:=&n;
new   3: n int:=5;
1
2
3
4
5
PL/SQL procedure successfully completed.
```

For Loop

FOR-LOOP:

Whereas the number of iterations through a WHILE loop is unknown until the loop completes, the number of iterations through a FOR loop is known before the loop is entered. FOR loops iterate over a specified range of integers. The range is part of an *iteration scheme*, which is enclosed by the keywords FOR and LOOP. A double dot (..) serves as the range operator. The syntax follows:

Syntax:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
  sequence_of_statements
END LOOP;
```

The range is evaluated when the FOR loop is first entered and is never re-evaluated.

By default, iteration proceeds upward from the lower bound to the higher bound. However, as the example below shows, if you use the keyword REVERSE, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. Nevertheless, you write the range bounds in ascending (not descending) order.

Syntax:

```
FOR i IN REVERSE 1..3 LOOP  -- assign the values 3,2,1 to i
  sequence_of_statements  -- executes three times
END LOOP;
```

Write a PL/SQL program to print sequence of n numbers using for loop?

```
SQL> declare
  2  a int:=1;
  3  n int:=&n;
  4  begin
  5  for a in 1..n
  6  loop
  7  dbms_output.put_line(a);
  8  end loop;
  9  end;
 10  /
Enter value for n: 5
old   3: n int:=&n;
new   3: n int:=5;
```

```
1
2
3
4
5
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print the sequence of n numbers in reverse using for loop?

```
SQL> declare
  2  n int:=&n;
  3  begin
  4  for a in reverse 1..n
  5  loop
  6  dbms_output.put_line(a);
  7  end loop;
  8  end;
  9  /
Enter value for n: 5
old   2: n int:=&n;
new   2: n int:=5;
5
4
3
2
1
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print a sequence of n numbers separated by spaces?

```
SQL> declare
  2  a int:=1;
  3  n int:=&n;
  4  begin
  5  loop
  6  dbms_output.put(a||' ');
  7  a:=a+1;
  8  exit when a>n;
  9  end loop;
 10  dbms_output.put_line('');
 11  end;
 12  /
Enter value for n: 5
old   3: n int:=&n;
new   3: n int:=5;
1 2 3 4 5
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to calculate factorial of a given number?

```
SQL> declare
  2  a int:=1;
  3  n int:=&n;
```

```
  4  f int:=n;
  5  begin
  6  while(n>=1)loop
  7  a:=a*n;
  8  n:=n-1;
  9  end loop;
 10  dbms_output.put_line('Factorial of '||f||' is '||a);
 11  end;
 12  /
Enter value for n: 5
old   3: n int:=&n;
new   3: n int:=5;
Factorial of 5 is 120
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print multiplication table?

```
SQL> declare
  2  a int:=1;
  3  n int:=&n;
  4  begin
  5  dbms_output.put_line('Multiplication Table:');
  6  while(a<=10)loop
  7  dbms_output.put_line(n||' x '||a||' = '||n*a);
  8  a:=a+1;
  9  end loop;
 10  end;
 11  /
Enter value for n: 2
old   3: n int:=&n;
new   3: n int:=2;
Multiplication Table:
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print Fibonacci Series?

```
SQL> declare
  2  a int:=0;
  3  b int:=1;
  4  c int:=1;
  5  n int:=&n;
  6  begin
```

```
 7  dbms_output.put_line('Fibonacci Series upto '||n);
 8  if(n=1) then
 9  dbms_output.put_line(a);
10  else
11  dbms_output.put(a||' '||b);
12  for x in 2..n-1
13  loop
14  dbms_output.put(' '||c);
15  a:=b;
16  b:=c;
17  c:=a+b;
18  end loop;
19  dbms_output.put_line('');
20  end if;
21  end;
22  /
Enter value for n: 10
old   5: n int:=&n;
new   5: n int:=10;
Fibonacci Series upto 10
0 1 1 2 3 5 8 13 21 34
PL/SQL procedure successfully completed.
```

Write a PL/SQL program to find whether the given number is prime or not?

```
SQL> declare
 2  n int:=&n;
 3  c int:=0;
 4  begin
 5  if(n<2) then
 6  dbms_output.put_line(n||' is not a prime number');
 7  else
 8  for i in 2..n-1
 9  loop
10  if mod(n,i)=0 then
11  c:=c+1;
12  end if;
13  end loop;
14  if(c=0) then
15  dbms_output.put_line(n||' is a prime number');
16  else
17  dbms_output.put_line(n||' is not a prime number');
18  end if;
19  end if;
20  end;
21  /
Enter value for n: 4
old   2: n int:=&n;
new   2: n int:=4;
4 is not a prime number
PL/SQL procedure successfully completed.
```

Experiment-11: Write a PL/SQL Code using Cursors, Exceptions and Composite Data Types.

CURSORS

A Cursor is a pointer to this context area. Oracle creates context area for processing an SQL statement which contains all information about the statement. The explicit cursor should be defined in the declaration section of the PL/SQL block, and it is created for the 'SELECT' statement that needs to be used in the code.

Steps:

1) Declaring the cursor
2) Opening Cursor
3) Fetching Data from the Cursor
4) Closing the Cursor

Syntax:

```
DECLARE
CURSOR <cursor_name> IS <SELECT statement^>
<cursor_variable declaration>
BEGIN
OPEN <cursor_name>;
FETCH <cursor_name> INTO <cursor_variable>;
.
.
CLOSE <cursor_name>;
END;
```

Cursor	Attributes
%FOUND	It returns the Boolean result 'TRUE' if the most recent fetch operation fetched a record successfully, else it will return FALSE.
%NOTFOUND	This works oppositely to %FOUND it will return 'TRUE' if the most recent fetch operation could not able to fetch any record.
%ROWTYPE	The attribute provides a record type that represents a row in a database table.

Write an explicit cursor using a simple loop?

```
SQL> declare
  2  cursor sail_cur is
  3  select sid,sname from sailors;
  4  ab sail_cur%rowtype;
  5  begin
  6  open sail_cur;
  7  loop
  8  fetch sail_cur into ab;
  9  exit when sail_cur%NOTFOUND;
 10  dbms_output.put_line(ab.sid||' '||ab.sname);
 11  end loop;
 12  close sail_cur;
 13  end;
 14  /
22 Dustin
```

29 Brutus
31 Lubber
32 Andy
64 Horatio
71 Zobra
74 Ravi
85 Art
95 Bob
58 Rusty

PL/SQL procedure successfully completed.

Write an explicit cursor using a while loop?

```
SQL> declare
  2  cursor sail_cur is
  3  select sid,sname from sailors;
  4  ab sail_cur%rowtype;
  5  begin
  6  open sail_cur;
  7  fetch sail_cur into ab;
  8  while sail_cur%FOUND
  9  loop
 10  dbms_output.put_line(ab.sid||' '||ab.sname);
 11  fetch sail_cur into ab;
 12  end loop;
 13  close sail_cur;
 14  end;
 15  /
```

22 Dustin
29 Brutus
31 Lubber
32 Andy
64 Horatio
71 Zobra
74 Ravi
85 Art
95 Bob
58 Rusty

PL/SQL procedure successfully completed.

Write an explicit cursor using for loop?

```
SQL> declare
  2  cursor sail_cur is
  3  select sid,sname from sailors;
  4  ab sail_cur%rowtype;
  5  begin
  6  for ab in sail_cur
  7  loop
  8  dbms_output.put_line(ab.sid||' '||ab.sname);
```

```
 9  end loop;
10  end;
11  /
22 Dustin
29 Brutus
31 Lubber
32 Andy
64 Horatio
71 Zobra
74 Ravi
85 Art
95 Bob
58 Rusty

PL/SQL procedure successfully completed.
```

EXCEPTIONS

An error occurs during the program execution is called Exception.
Two types of Exceptions

1) Predefined Exceptions
2) Userdefined Exceptions

Syntax:

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling goes here >
  WHEN exception1 THEN
     exception1-handling-statements
  WHEN exception2  THEN
    exception2-handling-statements
  WHEN others THEN
    exception3-handling-statements
END;
```

Exception	**Oracle Error**	**Description**
CASE_NOT_FOUND	06592	It is raised when none of the choices in the "WHEN" clauses of a CASE statement is selected, and there is no else clause.
DUP_VAL_ON_INDEX	00001	It is raised when duplicate values are attempted to be stored in a column with unique index.
NO_DATA_FOUND	01403	It is raised when a select into statement returns no rows.
VALUE_ERROR	06502	It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs.
ZERO_DIVIDE	01476	It is raised when an attempt is made to divide a number by zero.

Write a PL/SQL program to print divide by zero exception?

WITHOUT EXCEPTION

```
SQL> declare
  2  id number:=12;
  3  begin
  4  id:=12/0;
  5  end;
  6  /
declare
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 4
```

WITH EXCEPTION

```
SQL> declare
  2  id number:=12;
  3  BEGIN
  4  id:=12/0;
  5  exception
  6  when zero_divide then
  7  dbms_output.put_line('Divide by Zero');
  8  end;
  9  /
Divide by Zero

PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print value error exception?

WITHOUT EXCEPTION

```
SQL> declare
  2  num number:=&num;
  3  begin
  4  dbms_output.put_line('Square root of '||num||' is '||sqrt(num));
  5  end;
  6  /
Enter value for num: -25
old   2: num number:=&num;
new   2: num number:=-25;
declare
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 4
```

WITH EXCEPTION

```
SQL> declare
  2  num number:=&num;
  3  begin
  4  dbms_output.put_line('Square root of '||num||' is '||sqrt(num));
```

```
  5  exception
  6  when value_error then
  7  dbms_output.put_line('Value Error');
  8  end;
  9  /
Enter value for num: -25
old   2: num number:=&num;
new   2: num number:=-25;
Value Error

PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print unique constraint violated exception?
WITHOUT EXCEPTION

```
SQL> declare
  2  roll_no int:=502;
  3  sname varchar2(20):='Alekhya';
  4  marks number:=99;
  5  begin
  6  insert into student values(roll_no,sname,marks);
  7  end;
  8  /
declare
*
ERROR at line 1:
ORA-00001: unique constraint (CSE215G6.SYS_C0010795) violated
ORA-06512: at line 6
```

WITH EXCEPTION

```
SQL> declare
  2  roll_no int:=502;
  3  sname varchar2(20):='Alekhya';
  4  marks number:=99;
  5  begin
  6  insert into student values(roll_no,sname,marks);
  7  exception
  8  when dup_val_on_index then
  9  dbms_output.put_line('Unique constraint violated!');
 10  end;
 11  /
Unique constraint violated!

PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print no data found exception?
WITHOUT EXCEPTION

```
SQL> declare
  2  id int:=&id;
  3  name varchar2(20);
  4  begin
```

```
  5  select sname into name from student where roll_no=id;
  6  dbms_output.put_line(name);
  7  end;
  8  /
Enter value for id: 501
old   2: id int:=&id;
new   2: id int:=501;
declare
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5
```

WITH EXCEPTION

```
SQL> declare
  2  id int:=&id;
  3  name varchar2(20);
  4  begin
  5  select sname into name from student where roll_no=id;
  6  dbms_output.put_line(name);
  7  exception
  8  when no_data_found then
  9  dbms_output.put_line('No data found');
 10  end;
 11  /
Enter value for id: 501
old   2: id int:=&id;
new   2: id int:=501;
No data found

PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print case not found exception?

WITHOUT EXCEPTION

```
SQL> declare
  2  grade char(1):='&grade';
  3  begin
  4  case grade
  5  when 'A' then dbms_output.put_line('Excellent');
  6  when 'B' then dbms_output.put_line('Good');
  7  when 'C' then dbms_output.put_line('Average');
  8  when 'D' then dbms_output.put_line('Poor');
  9  end case;
 10  end;
 11  /
Enter value for grade: E
old   2: grade char(1):='&grade';
new   2: grade char(1):='E';
declare
*
```

ERROR at line 1:
ORA-06592: CASE not found while executing CASE statement
ORA-06512: at line 4

WITH EXCEPTION

```
SQL> declare
  2  grade char(1):='&grade';
  3  begin
  4  case grade
  5  when 'A' then dbms_output.put_line('Excellent');
  6  when 'B' then dbms_output.put_line('Good');
  7  when 'C' then dbms_output.put_line('Average');
  8  when 'D' then dbms_output.put_line('Poor');
  9  end case;
 10  exception
 11  when case_not_found then
 12  dbms_output.put_line('Case not found');
 13  end;
 14  /
Enter value for grade: E
old   2: grade char(1):='&grade';
new   2: grade char(1):='E';
Case not found

PL/SQL procedure successfully completed.
```

COMPOSITE DATA TYPES

1) Table Type: Table is a collection in which the size of the array is not fixed. It has the numeric subscript type.

Syntax:

```
TYPE <tvpe name> IS TABLE OF <DATA TYPE>;
```

2) Associate array: An associative array is single-dimensional. It means that an associative array has a single column of data in each row, which is similar to a one-dimension array.

Syntax:

```
TYPE <type_name>
 IS TABLE OF <data type>
INDEX BY <data type>;
```

3) Record type: A Record type is a complex data type which allows the programmer to create a new data type with the desired column structure.

Syntax:

```
type <typename> is record
(
Columnname datatype;
);
```

4) Varray: Varray is a collection method in which the size of the array is fixed. The array size cannot be exceeded than its fixed value.

Syntax:

```
TYPE <type_name> IS VARRAY (<SIZE>) OF <DATA_TYPE>;
```

Write a PL/SQL program to print the student name and marks using table type?

```
SQL> declare
  2  type namet is table of varchar2(20);
  3  type grades is table of integer;
  4  names namet;
  5  marks grades;
  6  total integer;
  7  begin
  8  names:=namet('Shah','Mike','Maddi','Alex','Peter');
  9  marks:=grades(92,87,98,97,78);
 10  total:=names.count;
 11  dbms_output.put_line('Total '||total||' students');
 12  for i in 1..total loop
 13  dbms_output.put_line('Student: '||names(i)||' marks: '||marks(i));
 14  end loop;
 15  end;
 16  /
Total 5 students
Student: Shah marks: 92
Student: Mike marks: 87
Student: Maddi marks: 98
Student: Alex marks: 97
Student: Peter marks: 78

PL/SQL procedure successfully completed.
```

Write a PL/SQL program to print student id and student name by using RECORD TYPE?

```
SQL> declare
  2  type t_name is record
  3  (
  4  sname student.sname%TYPE,roll_no student.roll_no%TYPE);
  5  r_name t_name; --name record
  6  n_emp_id student.roll_no%TYPE:=502;
  7  begin
  8  select sname,roll_no INTO r_name from student where roll_no=n_emp_id;
  9  dbms_output.put_line(r_name.sname||' , '||r_name.roll_no);
 10  end;
 11  /
Saketh , 502

PL/SQL procedure successfully completed.
```

Experiment-12: Write a PL/SQL Code using Procedures, Functions, Packages.

PROCEDURES

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.
The procedure contains a header and a body.

- Header: The header contains the name of the procedure and the parameters or variables passed to the procedure.
- Body: The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
   [ (parameter [,parameter]) ]
IS
   [declaration_section]
BEGIN
   executable_section
[EXCEPTION
   exception_section]
END [procedure_name];
```

Execute a Stored Procedure:

EXEC *procedure_name*;

Write a stored procedure to print hello message?

```
SQL> create procedure greet as
  2  begin
  3  dbms_output.put_line('Hello');
  4  end;
  5  /

Procedure created.

SQL> begin
  2  greet;
  3  end;
  4  /
Hello

PL/SQL procedure successfully completed.

SQL> exec greet;
Hello

PL/SQL procedure successfully completed.
```

Write a stored procedure to find addition of two numbers?

```
SQL> create procedure summing_c(a in number,b in number,c out number)
  2  as
  3  begin
  4  c:=a+b;
  5  end;
```

```
  6  /

Procedure created.

SQL> declare
  2  d number;
  3  begin
  4  summing_c(2,7,d);
  5  dbms_output.put_line(d);
  6  end;
  7  /
9

PL/SQL procedure successfully completed.
```

FUNCTIONS

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax to create a function:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
  < function_body >
END [function_name];
```

Here:

- Function_name: specifies the name of the function.
- [OR REPLACE] option allows modifying an existing function.
- The optional parameter list contains name, mode and types of the parameters.
- IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

The function must contain a return statement.

- RETURN clause specifies that data type you are going to return from the function.
- Function_body contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Write a stored function to find square of a given number?

```
SQL> create or replace function sq(x in number)
  2  return number
  3  as
  4  begin
  5  return (x*x);
  6  end;
  7  /

Function created.

SQL> begin
```

```
  2  dbms_output.put_line(sq(7));
  3  end;
  4  /
49

PL/SQL procedure successfully completed.

SQL> select sq(7) from dual;
    SQ(7)
----------
       49
```

Write a stored function to find addition of two numbers?

```
SQL> create or replace function add_c(a in number,b in number)
  2  return number
  3  as
  4  c number;
  5  begin
  6  c:=a+b;
  7  return c;
  8  end;
  9  /

Function created.

SQL> declare
  2  d number;
  3  begin
  4  d:=add_c(10,20);
  5  dbms_output.put_line(d);
  6  end;
  7  /
30

PL/SQL procedure successfully completed.
```

PACKAGES

A package is a way of logically storing the subprograms like procedures, functions, exception or cursor into a single common unit.
A package can be defined as an oracle object that is compiled and stored in the database. Once it is compiled and stored in the database it can be used by all the users of database who have executable permissions on Oracle database.
Components of Package
Package has two basic components:

- **Specification**: It is the declaration section of a Package
- **Body**: It is the definition section of a Package.

How to create a PL/SQL Package?
Following are the steps to declare and use a package in PL/SQL code block:

STEP 1: Package specification or declaration

It mainly comprises of the following:

- Package Name.
- Variable/constant/cursor/procedure/function/exception declaration.
- This declaration is global to the package.

Here is the syntax:

```
CREATE OR REPLACE PACKAGE <package_name> IS/AS
        FUNCTION <function_name> (<list of arguments>)
        RETURN <datatype>;
        PROCEDURE <procedure_name> (<list of arguments>);
        -- code statements
END <package_name>;
```

STEP 2: Package Body

It mainly comprises of the following:

- It contains the definition of procedure, function or cursor that is declared in the package specification.
- It contains the subprogram bodies containing executable statements for which package has been created

Here is the syntax:

```
CREATE  OR REPLACE PACKAGE BODY <package_name> IS/AS
FUNCTION <function_name> (<list of arguments>) RETURN <datatype>IS/AS
        -- local variable declaration;
        BEGIN
                -- executable  statements;
        EXCEPTION
                -- error handling statements;
END <function_name>;
PROCEDURE <procedure_name> (<list of arguments>)IS/AS
        -- local variable declaration;
        BEGIN
                -- executable statements;
        EXCEPTION
                -- error handling statements;
        END <procedure_name>;
END <package_name>;
```

Note: Creating a package only defines it, to use it we must refer it using the package object.
Following is the syntax for **referring a package object**:
Packagename.objectname;
The Object can be a function, procedure, cursor, exception that has been declared in the package specification.

Write a package to print hello message?

```
SQL> create package ss1
  2  as procedure greet;
  3  end;
  4  /

Package created.

SQL> create package body ss1
  2  as procedure greet as
  3  begin
  4  dbms_output.put_line('Hello!');
  5  end;
  6  end;
  7  /

Package body created.

SQL> exec ss1.greet;
Hello!

PL/SQL procedure successfully completed.
```

Write a package that prints student name by passing student id?

```
SQL> create or replace package pk as
  2  function fun1(no in number)
  3  return varchar2;
  4  end;
  5  /

Package created.

SQL> create or replace package body pk
  2  is
  3  function fun1(no in number) return varchar2
  4  is
  5  name varchar2(20);
  6  begin
  7  select sname into name from student where roll_no=no;
  8  return name;
  9  end;
 10  end;
 11  /

Package body created.

SQL> select pk.fun1(502) from dual;
PK.FUN1(502)
--------------------------------------------------------------------------------
Saketh
```