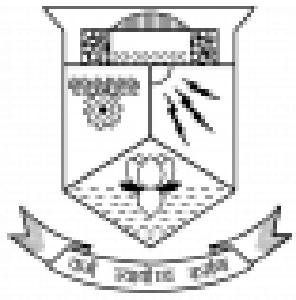


COLLEGE OF ENGINEERING TRIVANDRUM
DEPARTMENT OF COMPUTER SCIENCE

APPLICATION SOFTWARE DEVELOPMENT
LAB REPORT(CS333)

HISHAM ALI P

REG. NO : LTVE17CS066

ROLL NO : 67

S5 CSE

Staff In-charge:

VIPIN VASU A. V

Associate Professor

Department of Computer Science

Contents

1 Experiment 0
INSTALLATION OF POSTGRESQL

AIM:

To install PostgreSQL

PROCEDURE:

Step 1:Update system and install dependencies

sudo apt update

sudo apt install -y wget

```
hishamalip@savage-xubuntu:~$ sudo apt update && sudo apt install -y wget
[sudo] password for hishamalip:
Hit:1 http://in.archive.ubuntu.com/ubuntu bionic InRelease
Ign:2 http://dl.google.com/linux/chrome/deb stable InRelease
Hit:3 http://ppa.launchpad.net/noobslab/apps/ubuntu bionic InRelease
Hit:4 http://security.ubuntu.com/ubuntu bionic-security InRelease
Hit:5 http://in.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:6 http://dl.google.com/linux/chrome/deb stable Release
Hit:8 http://ppa.launchpad.net/teejee2008/ppa/ubuntu bionic InRelease
Get:9 http://in.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Hit:10 http://packages.microsoft.com/repos/vscode stable InRelease
Hit:11 http://ppa.launchpad.net/webupd8team/indicator-kdeconnect/ubuntu bionic I
nRelease
Fetched 74.6 kB in 8s (9,722 B/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
3 packages can be upgraded. Run 'apt list --upgradable' to see them.
Reading package lists... Done
Building dependency tree
Reading state information... Done
wget is already the newest version (1.19.4-1ubuntu2.2).
wget set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 3 not upgraded.
```

Step 2: Add PostgreSQL 11 APT repository

Download the key for postgresql repository and add it to the package manager keyring

wget –quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -

RELEASE=$$lsb/_release-cs$$

echo "deb http://apt.postgresql.org/pub/repos/apt/ $RELEASE"-pgdg main | sudo tee /etc/apt/sources.list.d/pgdg.list

cat /etc/apt/sources.list.d/pgdg.list

```
hishamalip@savage-xubuntu:~$ wget --quiet -O - https://www.postgresql.org/media/keys
/ACCC4CF8.asc | sudo apt-key add -
[sudo] password for hishamalip:
OK
hishamalip@savage-xubuntu:~$ RELEASE=$(lsb_release -cs)
hishamalip@savage-xubuntu:~$ echo "deb http://apt.postgresql.org/pub/repos/apt/ ${RE
LEASE}"-pgdg main | sudo tee /etc/apt/sources.list.d/pgdg.list
deb http://apt.postgresql.org/pub/repos/apt/ bionic-pgdg main
hishamalip@savage-xubuntu:~$ cat /etc/apt/sources.list.d/pgdg.list
deb http://apt.postgresql.org/pub/repos/apt/ bionic-pgdg main
hishamalip@savage-xubuntu:~$
```

Step 3: Install PostgreSQL 11 on Ubuntu 18.04 / Ubuntu 16.04

sudo apt update

sudo apt -y install postgresql-11

```
hishamalip@savage-xubuntu:~$ sudo apt update && sudo apt -y install postgresql-11
Ign:1 http://dl.google.com/linux/chrome/deb stable InRelease
Hit:2 http://in.archive.ubuntu.com/ubuntu bionic InRelease
Get:3 http://security.ubuntu.com/ubuntu bionic-security InRelease [88.7 kB]
Hit:4 http://dl.google.com/linux/chrome/deb stable Release
Hit:5 http://in.archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:7 http://packages.microsoft.com/repos/vscode stable InRelease
Get:8 http://security.ubuntu.com/ubuntu bionic-security/main i386 Packages [378 kB]
Get:9 http://in.archive.ubuntu.com/ubuntu bionic-backports InRelease [74.6 kB]
Hit:10 http://ppa.launchpad.net/noobslab/apps/ubuntu bionic InRelease
Get:11 http://security.ubuntu.com/ubuntu bionic-security/main amd64 Packages [523 kB]
Get:12 http://apt.postgresql.org/pub/repos/apt bionic-pgdg InRelease [46.3 kB]
Hit:13 http://ppa.launchpad.net/teejee2008/ppa/ubuntu bionic InRelease
Get:14 http://security.ubuntu.com/ubuntu bionic-security/main Translation-en [175 kB]
Hit:15 http://ppa.launchpad.net/webupd8team/indicator-kdeconnect/ubuntu bionic InRelease
Get:16 http://apt.postgresql.org/pub/repos/apt bionic-pgdg/main i386 Packages [159 kB]
Get:17 http://security.ubuntu.com/ubuntu bionic-security/universe i386 Packages [593 kB]
Get:18 http://apt.postgresql.org/pub/repos/apt bionic-pgdg/main amd64 Packages [160 kB]
Get:19 http://security.ubuntu.com/ubuntu bionic-security/universe amd64 Packages [609 kB]
```

Step 5: Post-Installation

Postgres has been successfully installed .You can now use postgres by following the steps given below

sudo su - postgres

psql

This will open the postgresql command line

```
hishamalip@savage-xubuntu:~$
hishamalip@savage-xubuntu:~$ sudo su - postgres
postgres@savage-xubuntu:~$ psql
psql (11.5 (Ubuntu 11.5-1.pgdg18.04+1))
Type "help" for help.

postgres=#
```

RESULT:

PostgreSQL installed in the PC and a database created. Postgresql version: 11.4 ,Ubuntu 18.04

2 Experiment 1
INTRODUCTION TO SQL

AIM:

Get to know about SQL and its queries

History of SQL

Dr. E. F. Codd published the paper, "A Relational Model of Data for Large Shared Data Banks", in June 1970 in the Association of Computer Machinery (ACM) journal, Communications of the ACM. Codd's model is now accepted as the definitive model for relational database management systems (RDBMS). The language, Structured English Query Language ("SEQUEL") was developed by IBM Corporation, Inc., to use Codd's model. SEQUEL later became SQL (still pronounced "sequel"). In 1979, Relational Software, Inc. (now Oracle Corporation) introduced the first commercially available implementation of SQL. Today, SQL is accepted as the standard RDBMS language.

How SQL Works

The strengths of SQL provide benefits for all types of users, including application programmers, database administrators, managers, and end users. Technically speaking, SQL is a data sub-language. The purpose of SQL is to provide an interface to a relational database such as Oracle, and all SQL statements are instructions to the database. In this SQL differs from general-purpose programming languages like C and BASIC. Among the features of SQL are the following:

1. It processes sets of data as groups rather than as individual units.
2. It provides automatic navigation to the data.
3. It uses statements that are complex and powerful individually, and that therefore stand alone.

Flow-control statements were not part of SQL originally, but they are found in the recently accepted optional part of SQL, ISO/IEC 9075-5: 1996. Flow-control statements are commonly known as "persistent stored modules" (PSM), and Oracle's PL/SQL extension to SQL is similar to PSM. Essentially, SQL lets you work with data at the logical level. You need to be concerned with the implementation details only when you want to manipulate the data. For example, to retrieve a set of rows from a table, you define a condition used to filter the rows. All rows satisfying the condition are retrieved in a single step and can be passed as a unit to the user, to another SQL statement, or to an

application. You need not deal with the rows one by one, nor do you have to worry about how they are physically stored or retrieved. All SQL statements use the optimizer, a part of Oracle that determines the most efficient means of accessing the specified data. Oracle also provides techniques you can use to make the optimizer perform its job better.

SQL provides statements for a variety of tasks, including:
1. Querying data
2. Inserting, updating, and deleting rows in a table
3. Creating, replacing, altering, and dropping objects
4. Controlling access to the database and its objects
5. Guaranteeing database consistency and integrity

SQL unifies all of the above tasks in one consistent language.
Common Language for All Relational Databases
All major relational database management systems support SQL, so you can transfer all skills you have gained with SQL from one database to another. In addition, all programs written in SQL are portable. They can often be moved from one database to another with very little modification.

Summary of SQL Statements
SQL statements are divided into these categories:
1. Data Definition Language (DDL) Statements
2. Data Manipulation Language (DML) Statements
3. Transaction Control Statements (TCL)
4. Session Control Statement
5. System Control Statement

DDL
DDL is short name of Data Definition Language, which deals with database schemas and descriptions, of how the data should reside in the database.
• CREATE – to create database and its objects like (table, index, views, store procedure, function and triggers)
• ALTER – alters the structure of the existing database
• DROP – delete objects from the database
• TRUNCATE – remove all records from a table, including all spaces allocated for the

records are removed
• COMMENT – add comments to the data dictionary
• RENAME – rename an object

DML
DML is short name of Data Manipulation Language which deals with data manipulation, and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE etc, and it is used to store, modify, retrieve, delete and update data in database.
• SELECT – retrieve data from the a database
• INSERT – insert data into a table
• UPDATE – updates existing data within a table
• DELETE – Delete all records from a database table
• MERGE – UPSERT operation (insert or update) • CALL – call a PL/SQL or Java subprogram
• EXPLAIN PLAN – interpretation of the data access path
• LOCK TABLE – concurrency Control

Managing Tables
A table is a data structure that holds data in a relational database. A table is composed of rows and columns. A table can represent a single entity that you want to track within your system. This type of a table would represent a list of the employees within your organization, or the orders placed for your company's products. A table can also represent a relationship between two entities. This type of a table could portray the association between employees and their job skills, or the relationship of products to orders. Within the tables, foreign keys are used torepresent relationships.
Creating Tables
To create a table, use the SQL command CREATETABLE. Syntax: CREATE TABLE<TABLENAME>
(<FIELDNAME><DATATYPE><[SIZE]>,.....)
Altering Tables
Alter a table in an Oracle database for any of the following reasons:
1. To add one or more new columns to the table
2. To add one or more integrity constraints to a table
3. To modify an existing column's definition (datatype, length, default value, and NOT-NULL integrity constraint)

4. To modify data block space usage parameters (PCTFREE, PCTUSED)
5. To modify transaction entry settings (INITRANS, MAXTRANS)
6. To modify storage parameters (NEXT, PCTINCREASE, etc.)
7. To enable or disable integrity constraints associated with the table
8. To drop integrity constraints associated with the table

When altering the column definitions of a table, you can only increase the length of an existing column, unless the table has no records. You can also decrease the length of a column in an empty table. For columns of data type CHAR, increasing the length of a column might be a time consuming operation that requires substantial additional storage, especially if the table contains many rows. This is because the CHAR value in each row must be blank-padded to satisfy the new column length.

If you change the datatype (for example, from VARCHAR2 to CHAR), then the data in the column does not change. However, the length of new CHAR columns might change, due to blank-padding requirements.

Altering a table has the following implications:

1. If a new column is added to a table, then the column is initially null. You can add a column with a NOTNULL constraint to a table only if the table does not contain any rows.
2. If a view or PL/SQL program unit depends on a base table, then the alteration of the base table might affect the dependent object, and always invalidates the dependent object.

Privileges Required to Alter a Table

To alter a table, the table must be contained in your schema, or you must have either the ALTER object privilege for the table or the ALTERANYTABLE system privilege.

Dropping Tables

Use the SQL command DROPTABLE to drop a table. For example, the following statement drops the EMPTAB table:

If the table that you are dropping contains any primary or unique keys referenced by foreign keys to other tables, and if you intend to drop the FOREIGNKEY constraints of the child tables, then include the CASCADE option in the DROPTABLE command.

Datatype

Use the NUMBER datatype to store real numbers in a fixed-point or floating-point format. Numbers using this data type are guaranteed to be portable among different Oracle

platforms, and offer up to 38 decimal digits of precision.
For numeric columns you can specify the column as a floating-point number:
Columnname NUMBER Or, you can specify a precision (total number of digits) and scale (number of digits to the right of the decimal point):
Columnname NUMBER (<precision>, <scale>)
Although not required, specifying the precision and scale for numeric fields provides extra integrity checking on input. If a precision is not specified, then the columnstores values as given. Table shows examples of how data would be stored using different scale factors.
Using the DATE Datatype
Use the DATE datatype to store point-in-time values (dates and times) in a table. The DATE datatype stores the century, year, month, day, hours, minutes, and seconds.
fields of seven bytes each, corresponding to century, year, month, day, hour, minute, and second

Date Format
For input and output of dates, the standard Oracle default date format is DD-MON-YY. For example: '13-NOV-92'
To change this default date format on an instance-wide basis, use the NLSDATEFORMAT parameter. To change the format during a session, use the ALTER SESSION statement. To enter dates that are not in the current default date format, use the TODATE function with a format mask.
For example:
TODATE ('November 13, 1992', 'MONTH DD, YYYY')
If the date format DD-MON-YY is used, then YY indicates the year in the 20th century (for example, 31-DEC-92 is December 31, 1992). If you want to indicate years in any century other than the 20 th century, then use a different format mask, as shown above.
Time Format
Time is stored in 24-hour format HH:MM:SS. By default, the time in a date field is 12:00:00 A.M. (midnight) if no time portion is entered. In a time-only entry, the date portion defaults to the first day of the current month. To enter the time portion of a date, use the TODATE function with a format mask indicating the time portion, as in:
INSERT INTO Birthdaystab (bname, bday)
VALUES ('ANNIE',TODATE('13-NOV-92 10:56 A.M.'
,'DD-MON-YY HH:MI A.M.'));
To compare dates that have time data, use the SQL function TRUNC if you want to

ignore the time component. Use the SQL function SYSDATE to return the system date and time. The FIXEDDATE initialization parameter allows you to set SYSDATE to a constant; this can be useful for testing.

RESULT:

Understood the basics of SQL and the queries.

3 Experiment 2
BASIC SQL QUERIES – I

AIM:

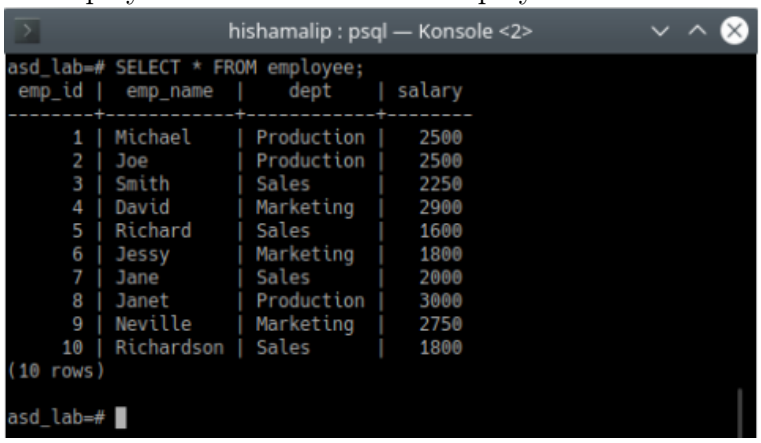
To study the basic sql queries such as

1. SELECT
2. INSERT
3. UPDATE
4. DELETE

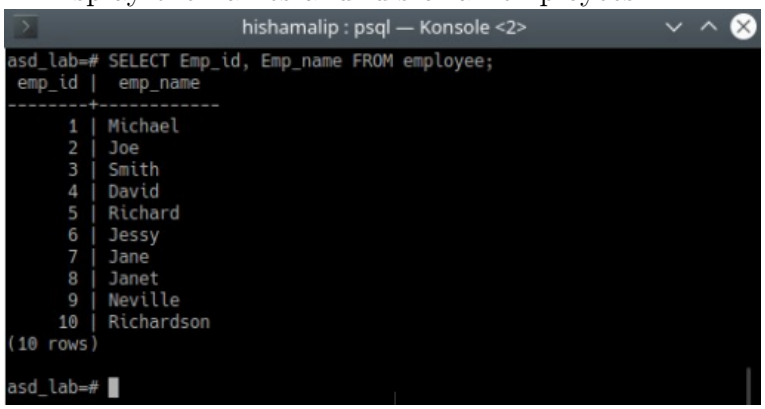
QUESTIONS:

Create a table named Employee and populate the table as shown below.

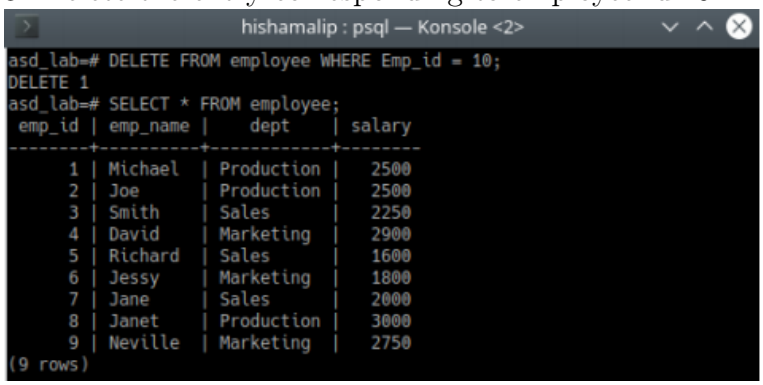
1. Display the details of all the employees.



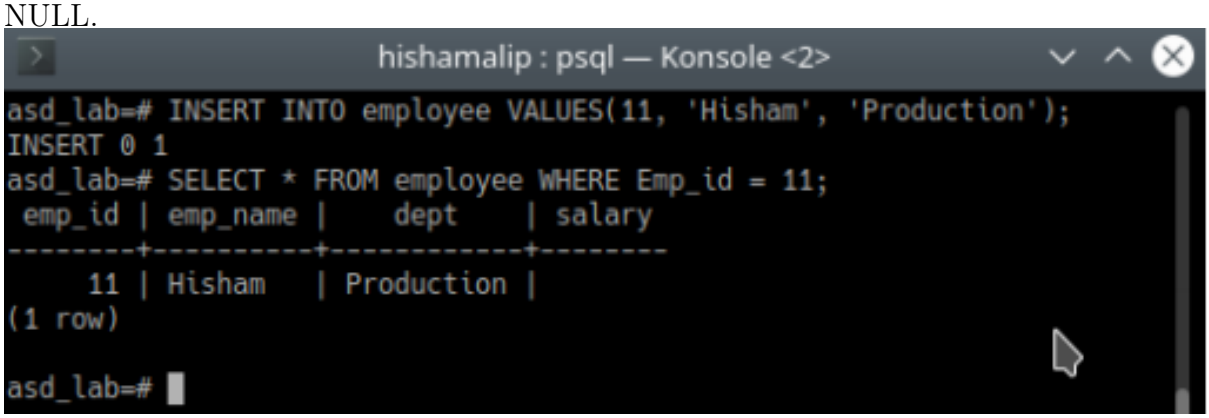
2. Display the names and id's of all employees.



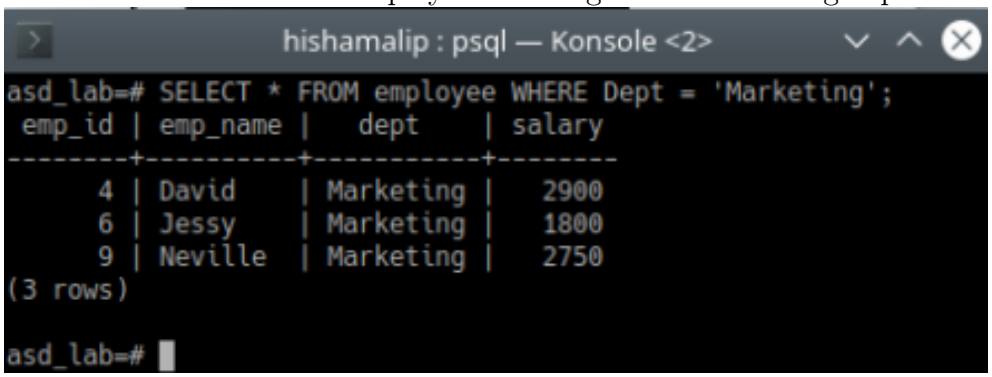
3. Delete the entry corresponding to employee id:10.



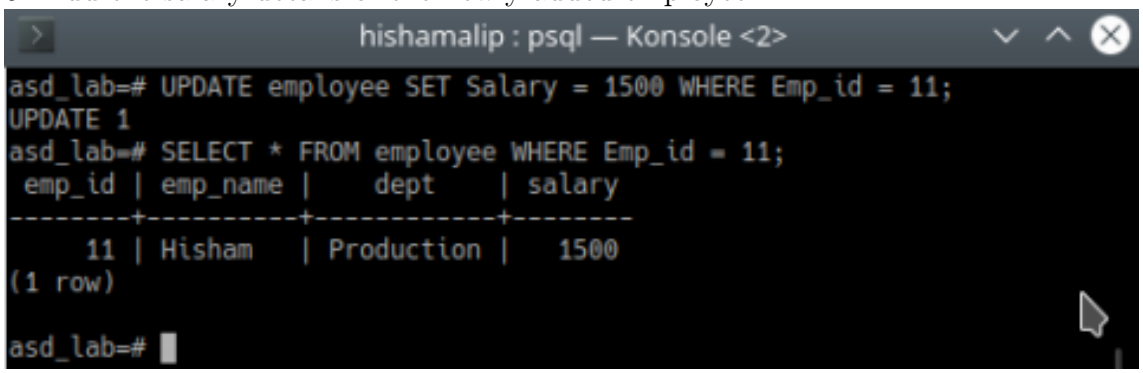
4. Insert a new tuple to the table. The salary field of the new employee should be kept NULL.



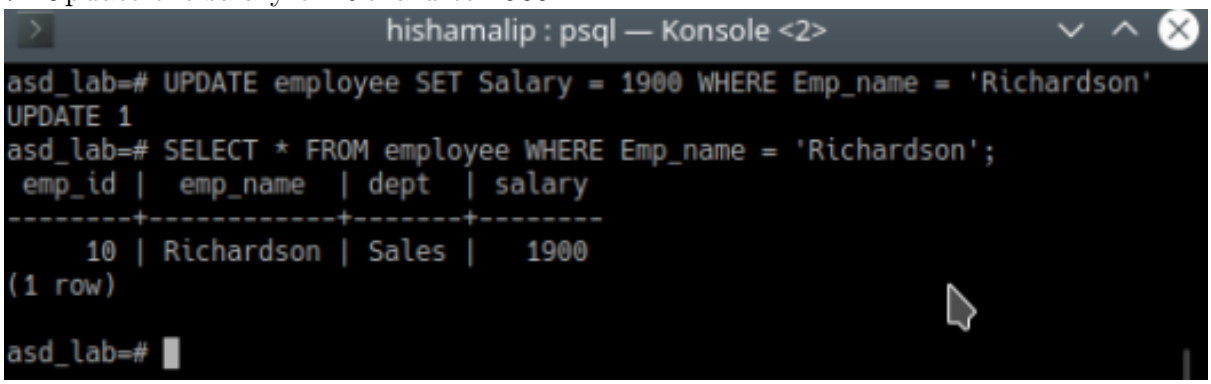
5. Find the details of all employees working in the marketing department



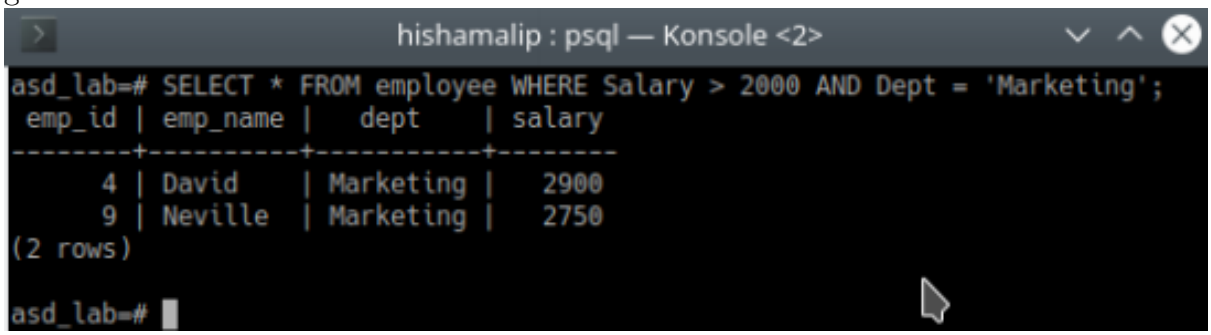
6. Add the salary details of the newly added employee



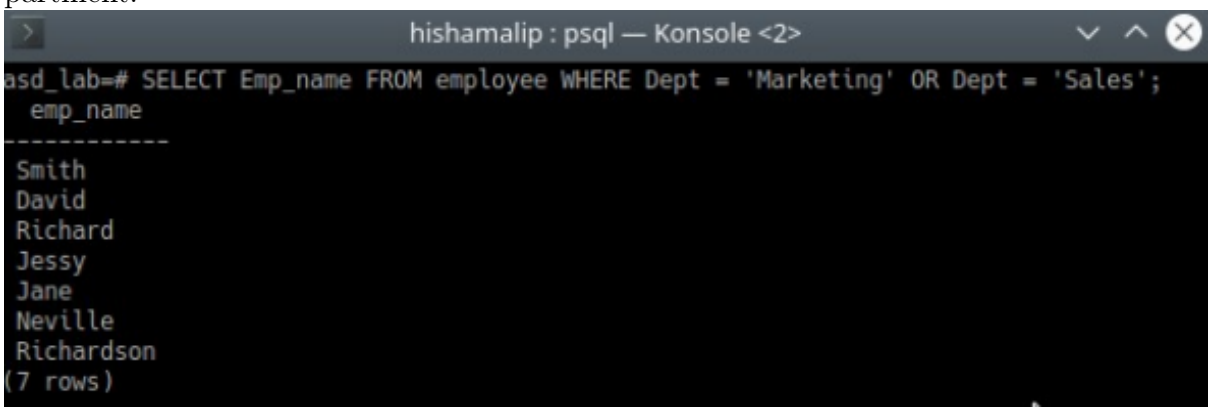
7. Update the salary of Richard to 1900



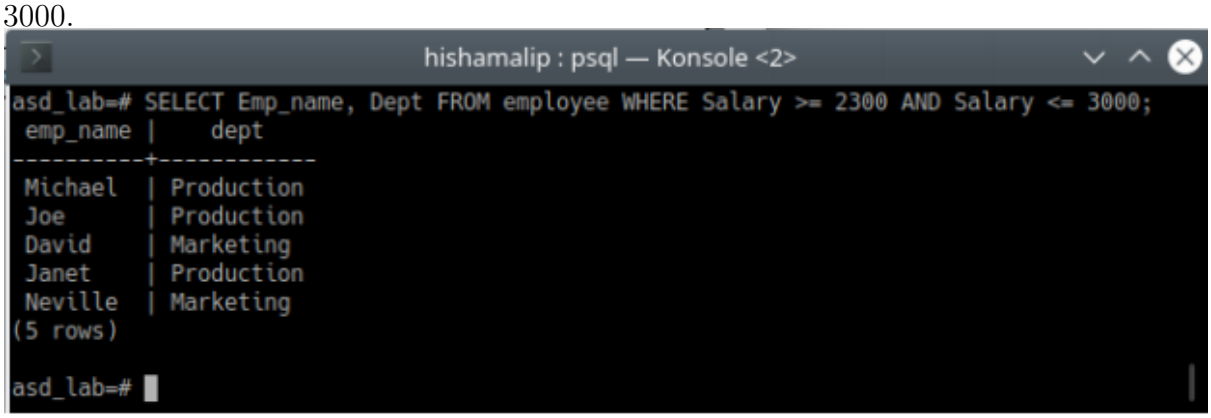
8. Find the details of all employees who are working for marketing and has a salary greater than 2000



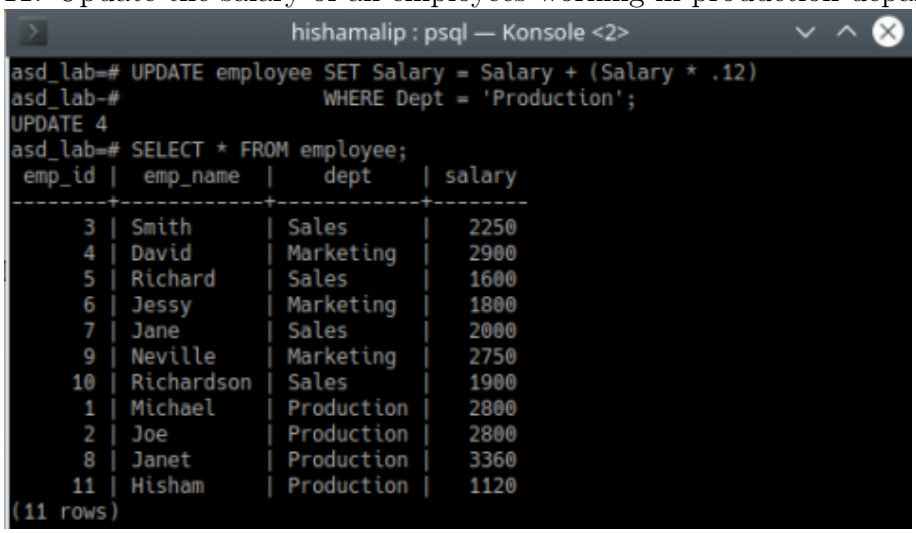
9. List the names of all employees working in the sales department and marketing department.



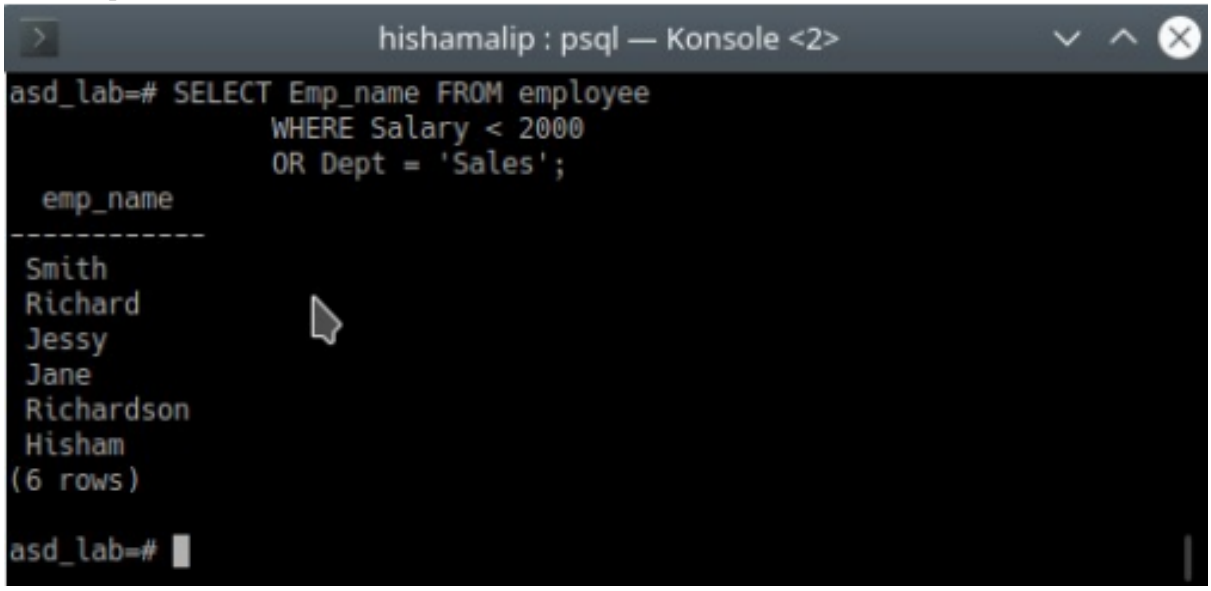
10. List the names and department of all employees whose salary is between2300 and 3000.



11. Update the salary of all employees working in production department 12%.



12. Display the names of all employees whose salary is less than 2000 or working for sales department.



RESULT:

Implemented the program for basic SQL queries-1 using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

4 Experiment 3
BASIC SQL QUERIES – II

AIM:

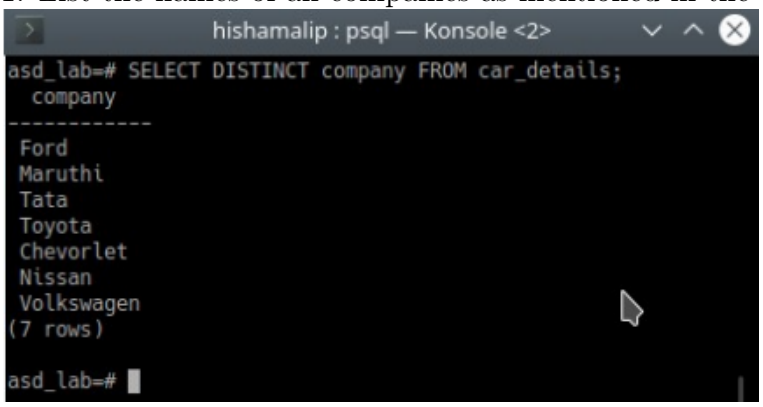
Introduction to SQL statements

1. ALTER
2. RENAME
3. SELECT DISTINCT
4. SQL IN
5. SQL BETWEEN
6. Sql aliases
7. Sql AND
8. Sql OR

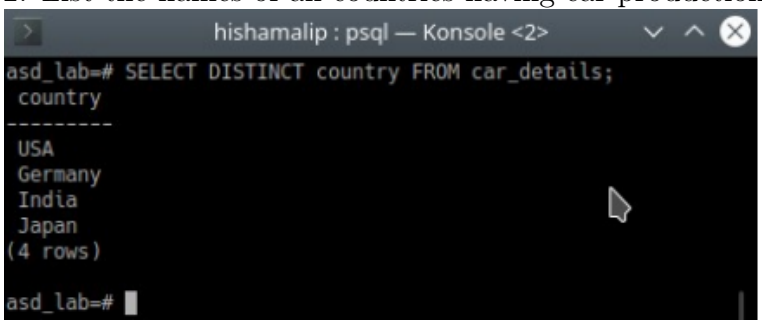
QUESTION:

Create a table named cardetails and populate the table as shown below.

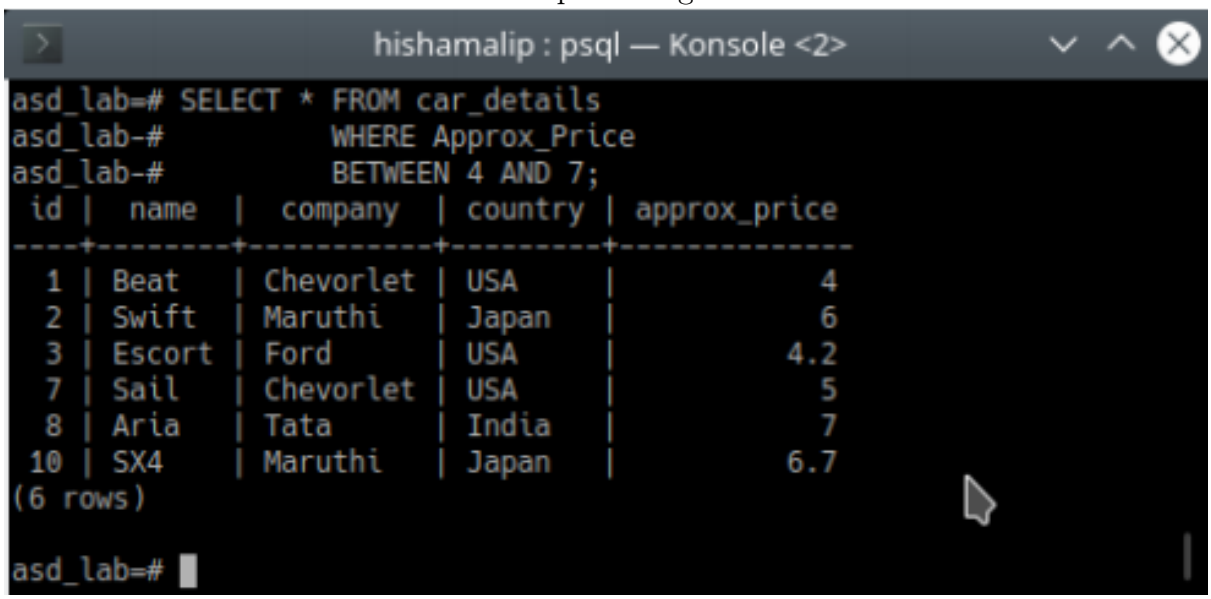
1. List the names of all companies as mentioned in the database



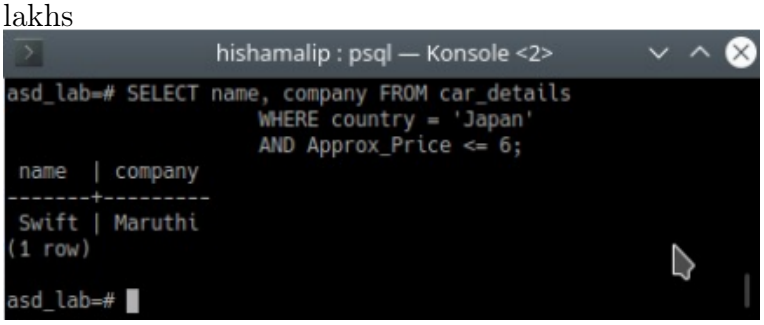
2. List the names of all countries having car production companies



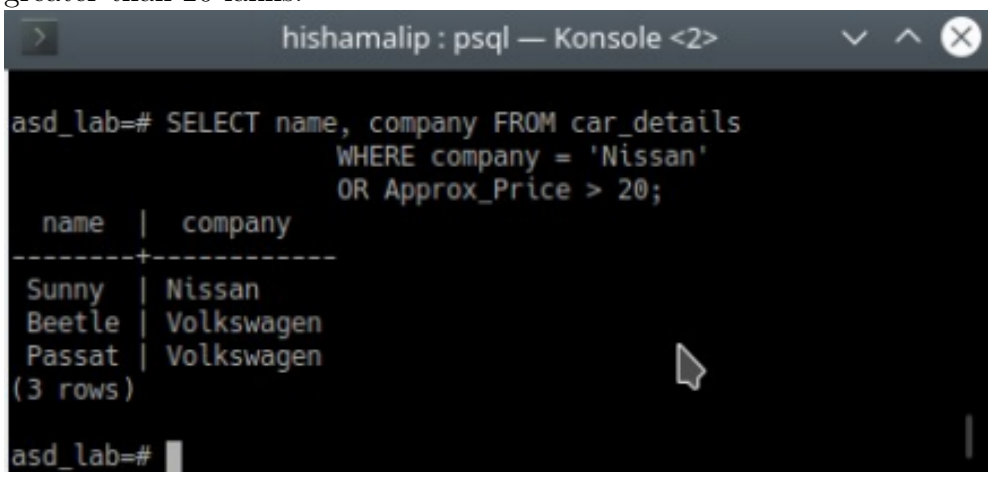
3.List the details of all cars within a price range 4 to 7 lakhs



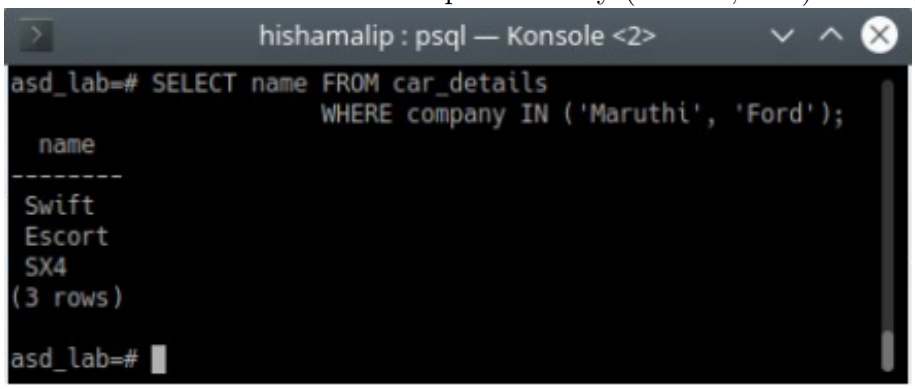
4. List the name and company of all cars originating from Japan and having price<=6 lakhs



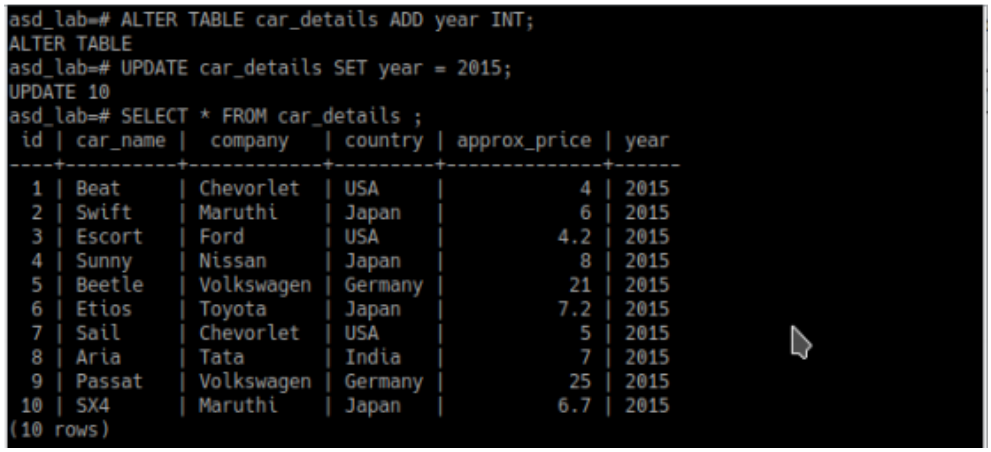
5. List the names and the companies of all cars either from Nissan or having a price greater than 20 lakhs.



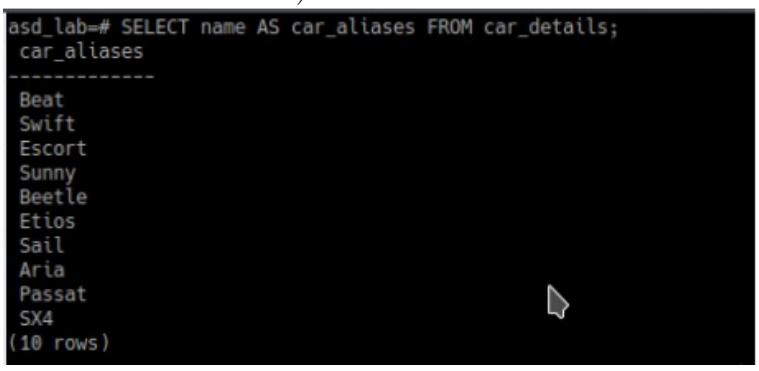
6. List the names of all cars produced by (Maruti,Ford).Use SQL IN statement.



7. Alter the table cars to add a new field year (model release year).Update the year column for all the rows in the database.



8. Display the names of all cars as Carname (while displaying the name attribute should be listed as caraliases)



9. Rename the attribute name to carname

```
asd_lab=# ALTER TABLE car_details RENAME name TO car_name;
ALTER TABLE
asd_lab=# select car_name from car_details ;
 car_name
----------
 Beat
 Swift
 Escort
 Sunny
 Beetle
 Etios
 Sail
 Aria
 Passat
 SX4
(10 rows)
```

10. List the car manufactured by Toyota(to be displayed as carsToyota)

```
asd_lab=# SELECT car_name AS cars_Toyota
asd_lab-#                 FROM car_details
asd_lab-#                 WHERE company = 'Toyota' ;
 cars_toyota
-------------
 Etios
(1 row)

asd_lab=#
```

11. List the details of all cars in alphabetical order

```
asd_lab=# SELECT * FROM car_details ORDER BY car_name ;
 id | car_name |  company   | country | approx_price | year
----+----------+------------+---------+--------------+------
  8 | Aria     | Tata       | India   |            7 | 2015
  1 | Beat     | Chevorlet  | USA     |            4 | 2015
  5 | Beetle   | Volkswagen | Germany |           21 | 2015
  3 | Escort   | Ford       | USA     |          4.2 | 2015
  6 | Etios    | Toyota     | Japan   |          7.2 | 2015
  9 | Passat   | Volkswagen | Germany |           25 | 2015
  7 | Sail     | Chevorlet  | USA     |            5 | 2015
  4 | Sunny    | Nissan     | Japan   |            8 | 2015
  2 | Swift    | Maruthi    | Japan   |            6 | 2015
 10 | SX4      | Maruthi    | Japan   |          6.7 | 2015
(10 rows)

asd_lab=#
```

12. List the details of all cars from cheapest to costliest.

```
asd_lab=# SELECT * FROM car_details ORDER BY approx_price ASC;
 id | car_name |  company   | country | approx_price | year
----+----------+------------+---------+--------------+------
  1 | Beat     | Chevorlet  | USA     |            4 | 2015
  3 | Escort   | Ford       | USA     |          4.2 | 2015
  7 | Sail     | Chevorlet  | USA     |            5 | 2015
  2 | Swift    | Maruthi    | Japan   |            6 | 2015
 10 | SX4      | Maruthi    | Japan   |          6.7 | 2015
  8 | Aria     | Tata       | India   |            7 | 2015
  6 | Etios    | Toyota     | Japan   |          7.2 | 2015
  4 | Sunny    | Nissan     | Japan   |            8 | 2015
  5 | Beetle   | Volkswagen | Germany |           21 | 2015
  9 | Passat   | Volkswagen | Germany |           25 | 2015
(10 rows)

asd_lab=#
```

RESULT:

Implemented the program for basic SQL queries-2 using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

5 Experiment 4
AGGREGATE FUNCTIONS

AIM:

Introduction to Aggregate functions
-AVG() -MAX() -MIN() - COUNT() -SUM()

THEORY:

- SUM(fieldname)
  Returns the total sum of the field.
- AVG(fieldname)
  Returns the average of the field.
- COUNT( )
  Count function has three variations:
  - COUNT(*) : returns the number of rows in the table including duplicates and those with null values
  - COUNT(fieldname) : returns the number of rows where field value is not null
  - COUNT (*): returns the total number of rows
- MAX(fieldname)
  Returns the maximum value of the field
- MIN(fieldname)
  Returns the maximum value of the field

QUESTION:

Create a table named student and populate the table as shown in the table. The table contains the marks of 10 students for 3 subjects(Physics, Chemistry, Mathematics).The total marks for physics and chemistry is 25, while for mathematics it is 50.The pass mark for physics and chemistry is 12 and for mathematics it is 25. A student is awarded a 'Pass' if he has passed all the subjects.

1. Find the class average for the subject 'Physics'.

```
asd_lab=# SELECT AVG(physics) FROM student;
         avg
---------------------
 16.2000000000000000
(1 row)

asd_lab=#
```

2. Find the highest marks for mathematics (To be displayed as highestmarksmaths).

```
hishamalip : psql — Konsole <2>
asd_lab=# SELECT MAX(maths) AS highest_marks_maths
asd_lab-#                   FROM student ;
 highest_marks_maths
---------------------
                  48
(1 row)

asd_lab=#
```

3. Find the lowest marks for chemistry(To be displayed as lowestmarkchemistry).

```
asd_lab=# SELECT MIN(chemistry) AS lowest_marks_chemistry
                 FROM student ;
 lowest_marks_chemistry
------------------------
                      7
(1 row)

asd_lab=#
```

4. Find the total number of students who has got a 'pass' in physics.

```
asd_lab=# SELECT COUNT(roll_no) FROM student
asd_lab-#                       WHERE physics >= 12;
 count
-------
     8
(1 row)

asd_lab=#
```

5. Generate the list of students who have passed in all the subjects.

```
asd_lab=# SELECT * FROM student
asd_lab-#          WHERE physics >=12
asd_lab-#          AND chemistry >= 12
asd_lab-#          AND maths >= 25 ;
 roll_no | name | physics | chemistry | maths
---------+------+---------+-----------+-------
       1 | Adam |      20 |        20 |    33
       8 | Mary |      24 |        14 |    31
(2 rows)
```

6. Generate a rank list for the class.Indicate Pass/Fail. Ranking based on total marks obtained by the students.

```
asd_lab=# ALTER TABLE student ADD COLUMN total_marks INT,
                              ADD COLUMN result TEXT;
ALTER TABLE
asd_lab=# UPDATE student SET total_marks = physics + chemistry + maths;
UPDATE 10
asd_lab=# UPDATE student SET result =
asd_lab-#                       CASE WHEN
asd_lab-#                           physics >= 12
asd_lab-#                           AND chemistry >= 12
asd_lab-#                           AND maths >= 25
asd_lab-#                       THEN 'P' ELSE 'F' END;
UPDATE 10
asd_lab=# SELECT * FROM student
asd_lab-#          ORDER BY total_marks DESC;
 roll_no |   name   | physics | chemistry | maths | total_marks | result
---------+----------+---------+-----------+-------+-------------+--------
       1 | Adam     |      20 |        20 |    33 |          73 | P
       8 | Mary     |      24 |        14 |    31 |          69 | P
       2 | Bob      |      18 |         9 |    41 |          68 | F
      10 | Zack     |       8 |        20 |    36 |          64 | F
       6 | Fletcher |       2 |        10 |    48 |          60 | F
       3 | Bright   |      22 |         7 |    31 |          60 | F
       5 | Elvin    |      14 |        22 |    23 |          59 | F
       9 | Tom      |      19 |        15 |    24 |          58 | F
       7 | Georgina |      22 |        12 |    22 |          56 | F
       4 | Duke     |      13 |        21 |    20 |          54 | F
(10 rows)
```

7. Find pass percentage of the class for mathematics.

```
asd_lab=# SELECT (
       (SELECT COUNT(*) FROM student WHERE maths >= 25)* 100)/ COUNT(*)
       AS maths_pass_percentage
       FROM student;
 maths_pass_percentage
-----------------------
                    60
(1 row)

asd_lab=#
```

8. Find the overall pass percentage for all class.

```
asd_lab=# SELECT (
       (SELECT COUNT(*) FROM student WHERE result = 'P')* 100)/ COUNT(*)
       AS pass_percentage
       FROM student;
 pass_percentage
-----------------
              20
(1 row)

asd_lab=#
```

9. Find the class average.

```
asd_lab=# SELECT SUM(total_marks)/ COUNT(*)
asd_lab-#        AS class_avg
asd_lab-#        FROM student ;
 class_avg
-----------
        62
(1 row)

asd_lab=#
```

10. Find the total number of students who have got a Pass.

```
asd_lab=# SELECT COUNT(*) FROM student
asd_lab-#                 WHERE result = 'P';
 count
-------
     2
(1 row)

asd_lab=#
```

RESULT:

Implemented the program for aggregate functions using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

6 Experiment 5
DATA CONSTRAINTS AND VIEWS

AIM:

To study about various data constraints and views in SQL.

THEORY:

Whenever two tables are related by a common column (or set of columns), define a PRIMARY or UNIQUE key constraint on the column in the parent table, and define a FOREIGNKEY constraint on the column in the child table, to maintain the relationship between the two tables.

When both UNIQUE and NOTNULL constraints are defined on the foreign key, only one row in the child table can reference a parent key value. Because nulls are not allowed in the foreign key, each row in the child table must explicitly reference a value in the parent key.

QUESTIONS:

1. Create the following tables with given constraints a. Create a table named Subjects with the given attributes * Subid(Should not be NULL) * Subname (Should not be NULL) Populate the database. Make sure that all constraints are working properly.

SUB_ID	SUB_NAME
1	Maths
2	Physics
3	Chemistry
4	English

Creating table and populating

```
CREATE TABLE
asd_lab=# INSERT INTO subjects VALUES(1,'maths'),
                                     (2, 'physics'),
                                     (3, 'chemistry'),
                                     (4, 'english');
INSERT 0 4
asd_lab=# SELECT * FROM subjects;
 subid |  subname
-------+-----------
     1 | maths
     2 | physics
     3 | chemistry
     4 | english
(4 rows)
```

i) Alter the table to set subid as the primary key.

ALTER TABLE subjects

ADD PRIMARY KEY (subid);

```
asd_lab=# ALTER TABLE subjects
asd_lab-#     ADD PRIMARY KEY (subid);
ALTER TABLE
asd_lab=#
```

b. Create a table named Staff with the given attributes -staffid (Should be UNIQUE)

-staffname

-dept

-Age (Greater than 22)

-Salary (Less than 35000)

Populate the database. Make sure that all constraints are working properly.

The staff TABLE has been succesfully created.

```
asd_lab=# CREATE TABLE staff(staffid INT UNIQUE,
                             staffname TEXT,
                             dept TEXT,
                             age INT CHECK (age > 22),
                             salary INT CHECK (salary < 35000));
CREATE TABLE
asd_lab=# INSERT INTO staff
asd_lab-# VALUES (1, 'John', 'Purchasing', 24, 30000),
asd_lab-#        (2, 'Sera', 'Sales', 25, 20000),
asd_lab-#        (3, 'Jane', 'Sales', 28, 25000);
INSERT 0 3
asd_lab=# SELECT * FROM staff;
 staffid | staffname |    dept    | age | salary
---------+-----------+------------+-----+--------
       1 | John      | Purchasing |  24 |  30000
       2 | Sera      | Sales      |  25 |  20000
       3 | Jane      | Sales      |  28 |  25000
(3 rows)
```

Check constraint is working for ageCheck, constraint is working for salary also. Unique Constraint is also working Properly. Populated Complete Table is given

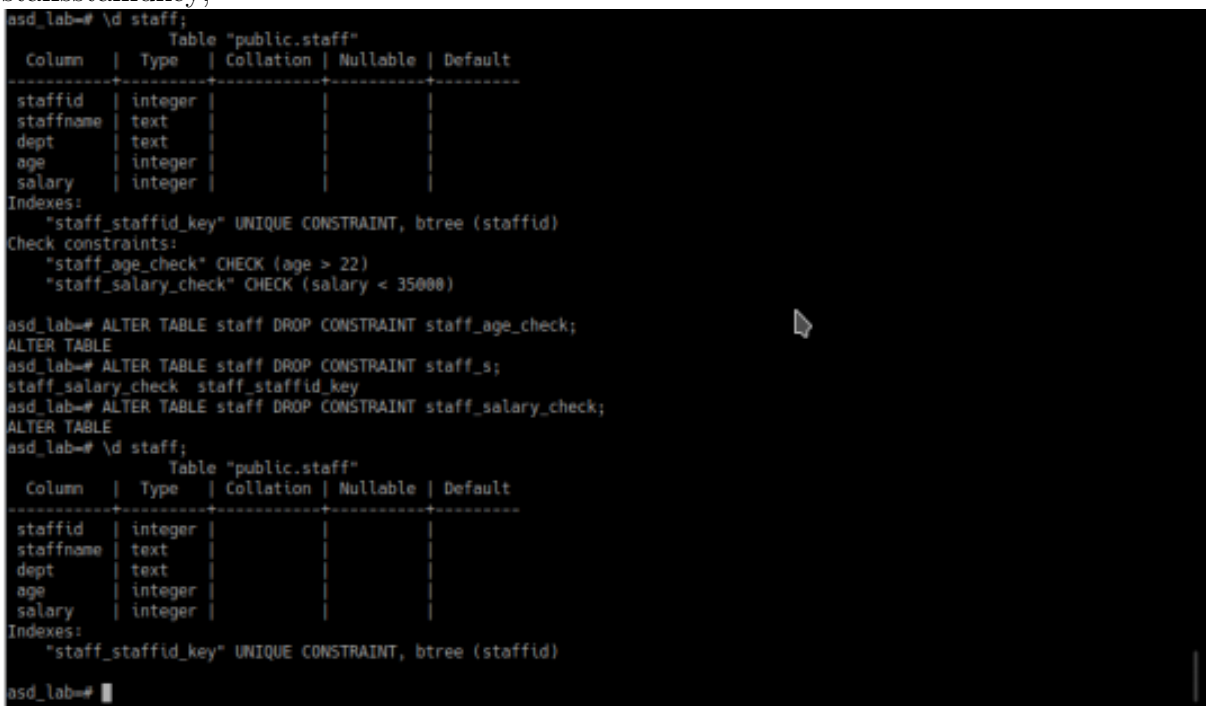
i)Delete the check constraint imposed on the attribute salary

ALTER TABLE staffs DROP CONSTRAINT chksalary;

We have added a salary of 35000 so that means that the Constraint has been removed

ii)Delete the unique constraint on the attribute staffid

ALTER TABLE staffs DROP CONSTRAINT staffsstaffidkey;



c. Create a table named Bank with the following attributes

-bankcode (To be set as Primary Key, type=varchar(3))

-bankname (Should not be NULL)

-headoffice-branches (Integer value greater than Zero)

Populate the database. Make sure that all constraints are working properly.All constraints have to be set after creating the table.

TABLE BANKING has been created

Primary Key constraint has been Added .

CHECK Constraint has been Added .

NOT NULL Constraint being Added.

Populated Table.

```
asd_lab=# CREATE TABLE bank(bankcode VARCHAR(3),
                  bankname VARCHAR(20),
                  headoffice VARCHAR(20),
                  branches INT);
CREATE TABLE
asd_lab=# ALTER TABLE bank
              ALTER COLUMN bankname SET NOT NULL;
ALTER TABLE
asd_lab=# ALTER TABLE bank ADD PRIMARY KEY(bankcode);
ALTER TABLE
asd_lab=# ALTER TABLE bank ADD CONSTRAINT branches CHECK(branches > 0);
ALTER TABLE
asd_lab=# \d bank
                      Table "public.bank"
   Column   |         Type          | Collation | Nullable | Default
------------+-----------------------+-----------+----------+---------
 bankcode   | character varying(3)  |           | not null |
 bankname   | character varying(20) |           | not null |
 headoffice | character varying(20) |           |          |
 branches   | integer               |           |          |
Indexes:
    "bank_pkey" PRIMARY KEY, btree (bankcode)
Check constraints:
    "branches" CHECK (branches > 0)

asd_lab=#
```

d. Create a table named Branch with the following attributes

-branchid (To be set as Primary Key)

-branchname (Set Default value as 'New Delhi')

-bankid (Foreign Key:- Refers to bank code of Bank table)

i) Populate the database. Make sure that all constraints are working properly

```
asd_lab=# CREATE TABLE branch(
                    branchid INT PRIMARY KEY,
                    branchname TEXT DEFAULT 'New Delhi',
                    bankid CHAR(3) REFERENCES bank(bankcode));
CREATE TABLE
asd_lab=# \d branch
                      Table "public.branch"
   Column   |     Type     | Collation | Nullable |      Default
------------+--------------+-----------+----------+-------------------
 branchid   | integer      |           | not null |
 branchname | text         |           |          | 'New Delhi'::text
 bankid     | character(3) |           |          |
Indexes:
    "branch_pkey" PRIMARY KEY, btree (branchid)
Foreign-key constraints:
    "branch_bankid_fkey" FOREIGN KEY (bankid) REFERENCES bank(bankcode)

asd_lab=#
```

ii) During database population, demonstrate how the DEFAULT Constraint is satisfied.

```
asd_lab=# INSERT INTO branch VALUES(01, 'Kottayam', 'CCC');
INSERT 0 1
asd_lab=# INSERT INTO branch(branchid, bankid) VALUES(02, 'AAA');
INSERT 0 1
asd_lab=# SELECT * FROM branch;
 branchid | branchname | bankid
----------+------------+--------
        1 | Kottayam   | CCC
        2 | New Delhi  | AAA
(2 rows)

asd_lab=#
```

iii) Delete the bank with bank code ‘SBT’ and make sure that the corresponding entries are getting deleted from the related tables.

```
asd_lab=# INSERT INTO bank VALUES('SBT', 'Indian', 'Delhi', 7);
INSERT 0 1
asd_lab=# INSERT INTO branch VALUES(5, 'Calicut', 'SBT');
INSERT 0 1
asd_lab=# SELECT * FROM bank ;
 bankcode | bankname | headoffice | branches
----------+----------+------------+----------
 AAA      | SIB      | Eranakulam |        6
 BBB      | Federal  | Kottayam   |        5
 CCC      | Canara   | Trivandrum |        3
 SBT      | Indian   | Delhi      |        7
(4 rows)

asd_lab=# SELECT * FROM branch;
 branchid | branchname | bankid
----------+------------+--------
        1 | Kottayam   | CCC
        2 | New Delhi  | AAA
        5 | Calicut    | SBT
(3 rows)

asd_lab=# DELETE FROM bank WHERE bankcode = 'SBT';
ERROR:  update or delete on table "bank" violates foreign key constraint "branch_bankid_
fkey" on table "branch"
DETAIL:  Key (bankcode)=(SBT) is still referenced from table "branch".
```

When trying to delete bankcode = ‘SBT’ from “bank” table, we got an error because it is still referenced from table “branch” . We can’t delete a foreign key if it still references another table.

To make the corresponding entries are getting delete from the related tables we need to add nON DELETE CASCADE to the existing foreign key. In order to do this drop the existing constraint and recreate it with addition of the ON DELETE clause

```
asd_lab=# ALTER TABLE branch
asd_lab-# DROP CONSTRAINT branch_bankid_fkey;
ALTER TABLE
asd_lab=#
asd_lab=# ALTER TABLE branch
asd_lab-#     ADD CONSTRAINT branch_bankid_fkey
asd_lab-#     FOREIGN KEY (bankid)
asd_lab-#     REFERENCES bank (bankcode)
asd_lab-#     ON DELETE CASCADE;
ALTER TABLE
asd_lab=#
```

After deletion

```
asd_lab=# DELETE FROM bank WHERE bankcode = 'SBT';
DELETE 1
asd_lab=# SELECT * FROM bank;
 bankcode | bankname | headoffice | branches
----------+----------+------------+----------
 AAA      | SIB      | Eranakulam |        6
 BBB      | Federal  | Kottayam   |        5
 CCC      | Canara   | Trivandrum |        3
(3 rows)

asd_lab=# SELECT * FROM branch;
 branchid | branchname | bankid
----------+------------+--------
        1 | Kottayam   | CCC
        2 | New Delhi  | AAA
(2 rows)
```

iv) Drop the Primary Key using ALTER command Dropping a Primary Key is done as ALTER TABLE BRANCH DROP CONSTRAINT branchpkey

Here branchpkey is the name by which the primary key of BRANCH can be seen .

we can say that the Primary Key has been Dropped

```
asd_lab=# \d branch
                    Table "public.branch"
   Column   |     Type     | Collation | Nullable |      Default
------------+--------------+-----------+----------+-------------------
 branchid   | integer      |           | not null |
 branchname | text         |           |          | 'New Delhi'::text
 bankid     | character(3) |           |          |
Indexes:
    "branch_pkey" PRIMARY KEY, btree (branchid)
Foreign-key constraints:
    "branch_bankid_fkey" FOREIGN KEY (bankid) REFERENCES bank(bankcode) ON DELETE CASCADE

asd_lab=# ALTER TABLE branch DROP CONSTRAINT branch_pkey ;
ALTER TABLE
asd_lab=# \d branch
                    Table "public.branch"
   Column   |     Type     | Collation | Nullable |      Default
------------+--------------+-----------+----------+-------------------
 branchid   | integer      |           | not null |
 branchname | text         |           |          | 'New Delhi'::text
 bankid     | character(3) |           |          |
Foreign-key constraints:
    "branch_bankid_fkey" FOREIGN KEY (bankid) REFERENCES bank(bankcode) ON DELETE CASCADE

asd_lab=#
```

VIEWS

1. Create a View named salesstaff to hold the details of all staff working in sales Department STAFF TABLE is given below

The VIEW salesstaff is given below

```
asd_lab=# CREATE VIEW sales_staff AS
asd_lab-#                           SELECT * FROM staff
asd_lab-#                           WHERE dept = 'Sales';
CREATE VIEW
asd_lab=# SELECT * FROM sales_staff;
 staffid | staffname | dept  | age | salary
---------+-----------+-------+-----+--------
       2 | Sera      | Sales |  25 |  20000
       3 | Jane      | Sales |  28 |  25000
(2 rows)
```

2. Drop table branch. Create another table named branch and name all the constraints as given below:

Constraint name Column Constraint

Pk branchid Primary key

Df branchname Default :'New Delhi'

Fk bankid Foreign key/References

```
asd_lab=# CREATE TABLE branch(
                  branch_id INT CONSTRAINT Pk PRIMARY KEY,
                  branch_name varchar(20) CONSTRAINT Df DEFAULT 'New Delhi',
                  bankid char(3) CONSTRAINT Fk REFERENCES bank(bankcode) ON DELETE CASCADE
                 );
CREATE TABLE
asd_lab=# \d branch
                            Table "public.branch"
   Column    |         Type          | Collation | Nullable |           Default
-------------+-----------------------+-----------+----------+-------------------------------
 branch_id   | integer               |           | not null |
 branch_name | character varying(20) |           |          | 'New Delhi'::character varying
 bankid      | character(3)          |           |          |
Indexes:
    "pk" PRIMARY KEY, btree (branch_id)
Foreign-key constraints:
    "fk" FOREIGN KEY (bankid) REFERENCES bank(bankcode) ON DELETE CASCADE

asd_lab=#
```

THE table BRANCH is dropped using the Statement

DROP TABLE BRANCH.

Here we Dropped the BRANCH Table and created a new One with the 3 Constraints Specified Outside

i) Delete the default constraint in the table

ii) Delete the primary key constraint

```
asd_lab=# ALTER TABLE branch
asd_lab-#     ALTER COLUMN branch_name DROP DEFAULT;
ALTER TABLE
asd_lab=#
asd_lab=# ALTER TABLE branch DROP CONSTRAINT Pk;
ALTER TABLE
asd_lab=# \d branch
                    Table "public.branch"
   Column    |         Type          | Collation | Nullable | Default
-------------+-----------------------+-----------+----------+---------
 branch_id   | integer               |           | not null |
 branch_name | character varying(20) |           |          |
 bankid      | character(3)          |           |          |
Foreign-key constraints:
    "fk" FOREIGN KEY (bankid) REFERENCES bank(bankcode) ON DELETE CASCADE

asd_lab=#
```

3. Update the view salesstaff to include the details of staff belonging to sales department whose salary is greater than 20000.A view can be updated with the CREATE OR REPLACE VIEW command.

CREATE OR REPLACE VIEW viewname AS

SELECT column1, column2, ...

FROM tablename

WHERE condition;

```
asd_lab=# CREATE OR REPLACE VIEW sales_staff AS
                                          SELECT * FROM staff
                                          WHERE dept = 'Sales'
                                                AND salary > 20000;
CREATE VIEW
asd_lab=# SELECT * FROM sales_staff ;
 staffid | staffname | dept  | age | salary
---------+-----------+-------+-----+--------
       3 | Jane      | Sales |  28 |  25000
(1 row)

asd_lab=#
```

4. Delete the view salesstaff.

A view is deleted with the DROP VIEW command.

DROP VIEW newsalesstuff;

```
asd_lab=# DROP VIEW sales_staff ;
DROP VIEW
asd_lab=# SELECT * FROM sales_staff ;
ERROR:  relation "sales_staff" does not exist
LINE 1: SELECT * FROM sales_staff ;
                      ^
asd_lab=#
```

RESULT:

Implemented the program for data-constraint and views using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

7 Experiment 6
STRING FUNCTIONS AND PATTERN MATCHING

AIM:

To understand

- SUBSTR - RPAD
- INITCAP - INSTR
- LPAD - CONCAT
- LTRIM - UPPER
- LENGTH - RTRIM
- LOWER - REVERSE

THEORY:

The main string functions are as follows

1.Length() 2.Lower() 3.Upper() 4.Concat()

5.Lpad() 6.Rpad() 7.Rtrim() 8.Substr()

-Length(field name/string)

Gives the length of the string.

-Lower(field name/string)

Gives the content in lowercase letters

-Upper(field name/string)

Gives the content in upper case letters

-Concat (field name/string, field name/string)

Combines the first and second string into single one.

-Lpad(field name/string,length,character)

QUESTIONS:

Create a table named acctdetails and populate the table

```
hishamalip@savage: ~
asdlab=# CREATE TABLE acct_detais(Acct_No TEXT, Branch TEXT, Name TEXT, Phone BIGINT);
CREATE TABLE
asdlab=# INSERT INTO acct_detais VALUES('A40123401', 'Chicago', 'Mike Adams', 3784001234),
                                       ('A40123402', 'Miami', 'Diana George', 3724202345),
                                       ('B40123403', 'Miami', 'Diaz Elizabeth', 3714503456),
                                       ('B40123404', 'Atlanta', 'Jeoffrey George', 3704604567),
                                       ('B40123405', 'New York', 'Jennifer Kaitlyn', 3734705678);
INSERT 0 5
asdlab=# INSERT INTO acct_detais VALUES('C40123406', 'Chicago', 'Kaitlyn Vincent', 3182003235),
                                       ('C40123407', 'Miami', 'Abraham Gottfield', 3283002256),
asdlab-#                               ('C50123408', 'New Jersey', 'Stacy Williams', 3384005237),
asdlab-#                               ('D50123409', 'New York', 'Catherine George', 3485006228),
asdlab-#                               ('D50123410', 'Miami', 'Oliver Scott', 3586007230);
INSERT 0 5
asdlab=#
```

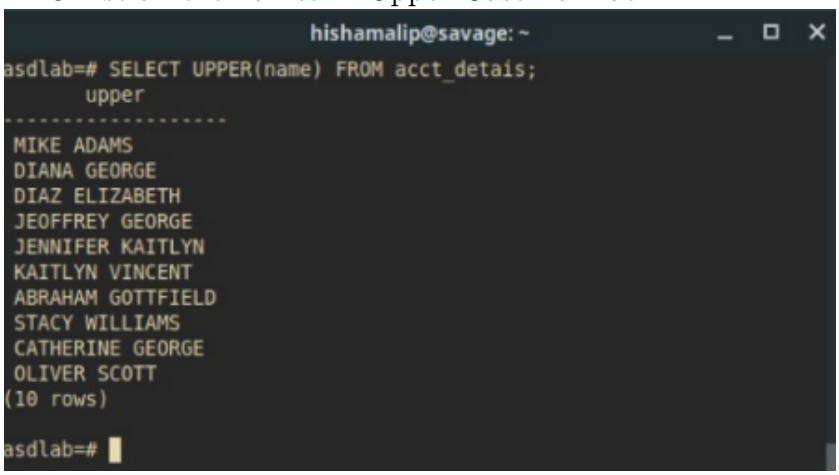
1.Find the names of all people starting on the alphabet 'D'

```
hishamalip@savage: ~
asdlab=# SELECT name FROM acct_detais WHERE Name LIKE 'D%';
      name
----------------
 Diana George
 Diaz Elizabeth
(2 rows)

asdlab=#
```

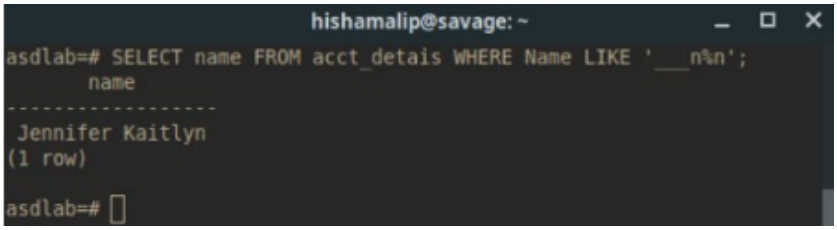
2.List the names of all branches containing the substring 'New'

```
hishamalip@savage: ~
asdlab=# SELECT branch FROM acct_detais WHERE branch LIKE '%New%';
   branch
------------
 New York
 New Jersey
 New York
(3 rows)

asdlab=#
```

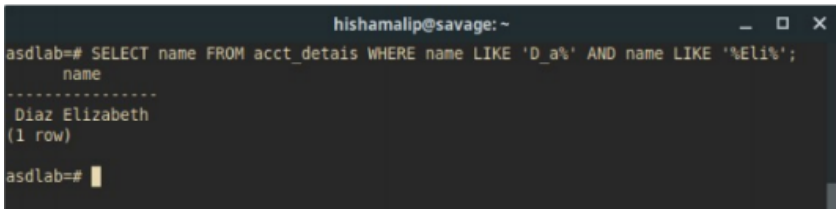
3.List all the names in Upper Case Format



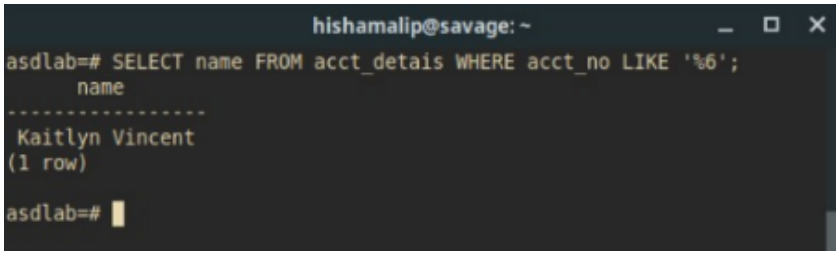
4.List the names where the 4th letter is ‘n’ and last letter is ‘n’



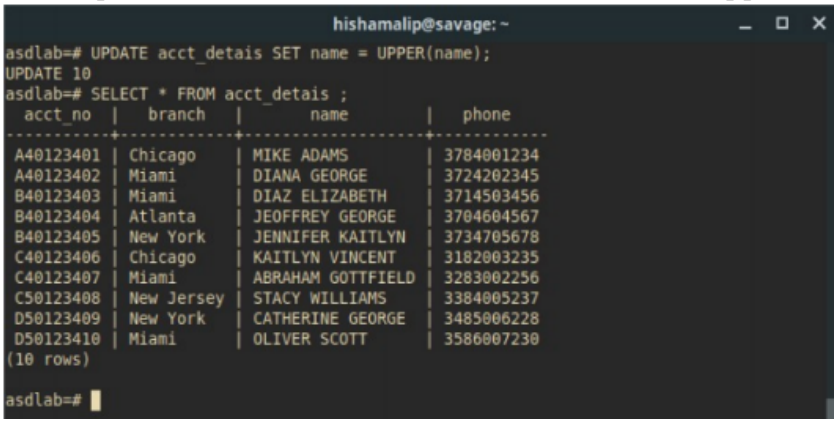
5.List the names starting on ‘D’ , 3rd letter is ‘a’ and contains the substring‘Eli’



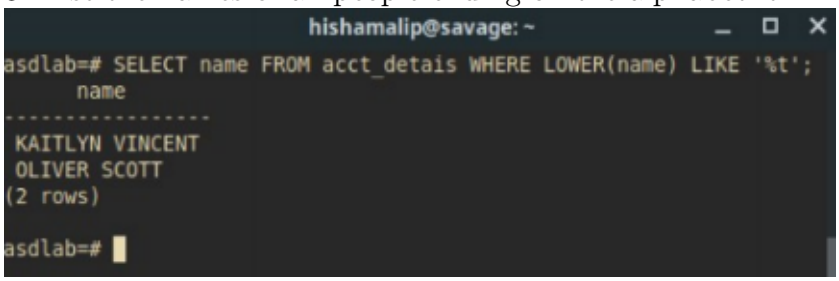
6. List the names of people whose account number ends in ‘6’



7.Update the table so that all the names are in Upper Case Format

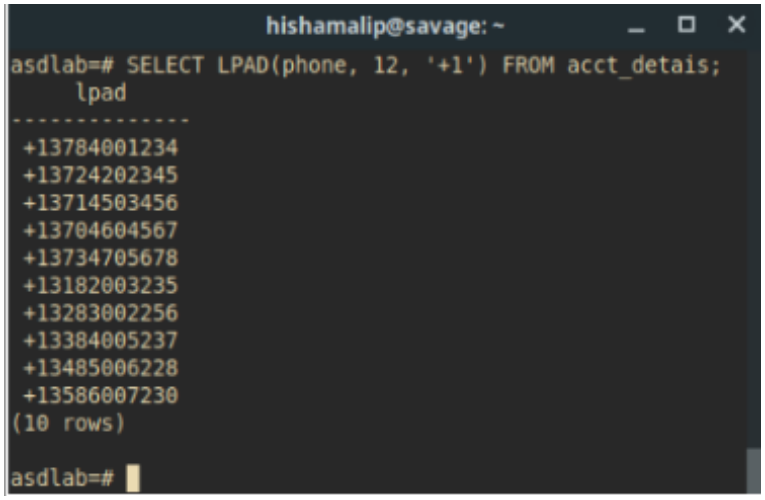


8. List the names of all people ending on the alphabet 't'

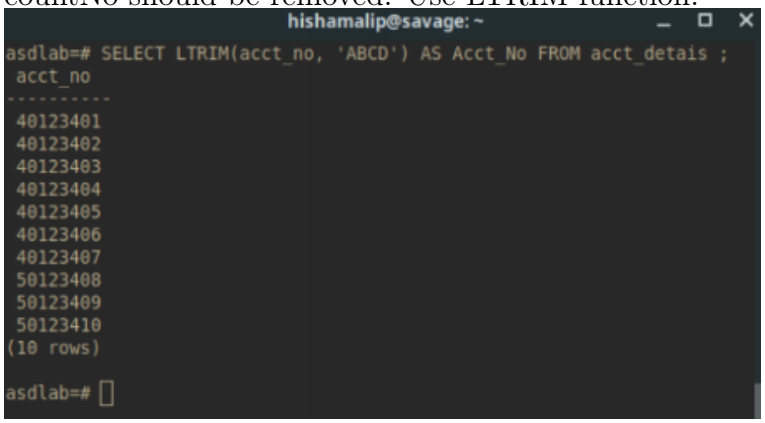


9.List all the names in reverse

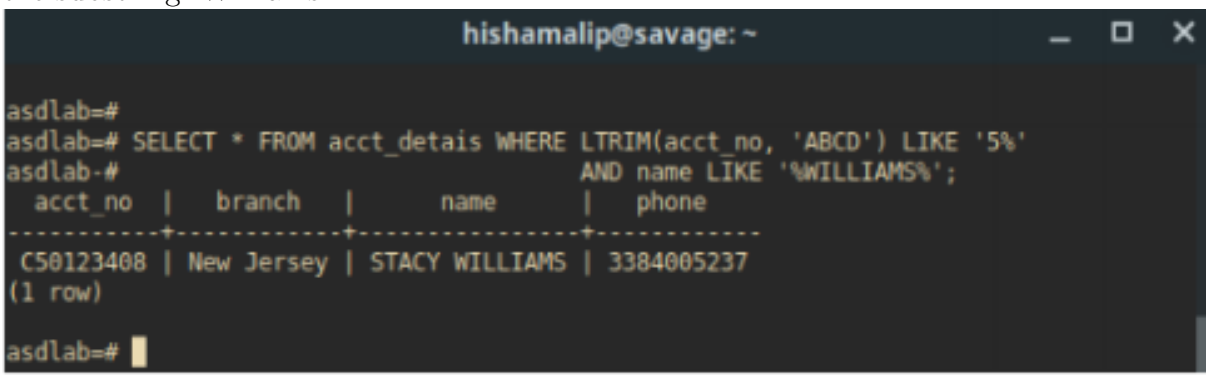
10. Display all the phone numbers including US Country code (+1). For eg: (378)400-1234should be displayed as +1(378)400-1234. Use LPAD function



11. Display all the account numbers. The starting alphabet associated with the AccountNo should be removed. Use LTRIM function.



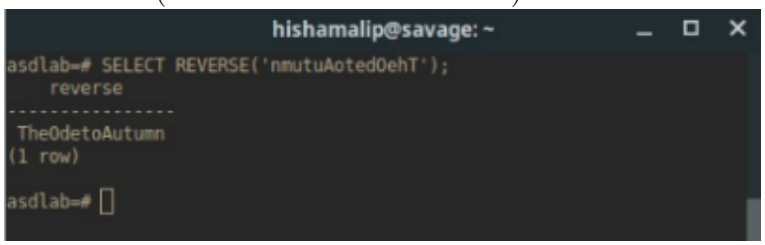
12. Display the details of all people whose account number starts in '4' and name contains the substring 'Williams'.



B.Use the system table DUAL for the following questions:

1.Find the reverse of the string ‘ nmutuAotedOehT’

REVERSE(’NMUTUAOTEDOEHT’)

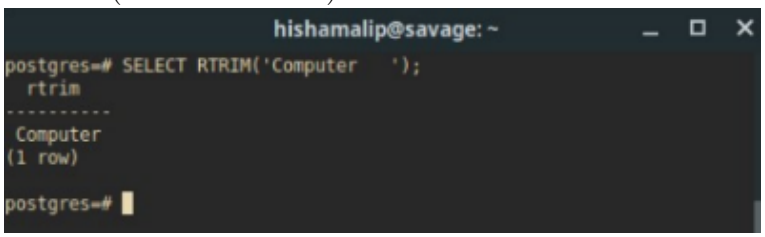


2.Use LTRIM function on ‘123231xyzTech’ so as to obtain the output ‘Tech’

LTRIM(’123231XYZTECH’,’123XYZ’)

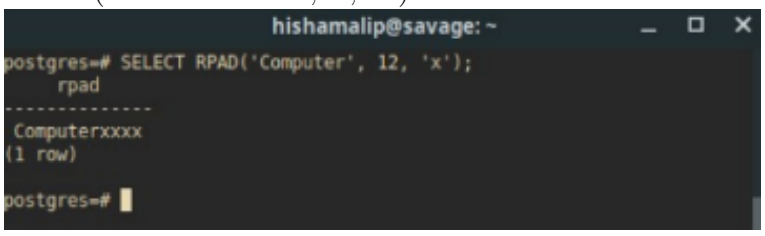
3.Use RTRIM function on ‘Computer ‘ to remove the trailing spaces.

RTRIM(’COMPUTER’)



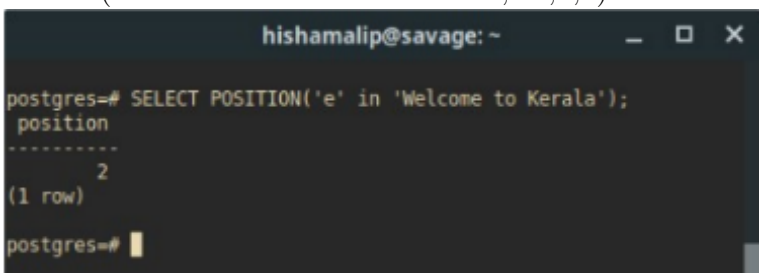
4.Perform RPAD on ‘computer’ to obtain the output as ‘computerXXXX’

RPAD(’COMPUTER’,12,’X’)



5.Use INSTR function to find the first occurrence of ‘e’ in the string ‘WelcometoKerala’

INSTR(’WELCOME TO KERALA’,’E’,1,1)

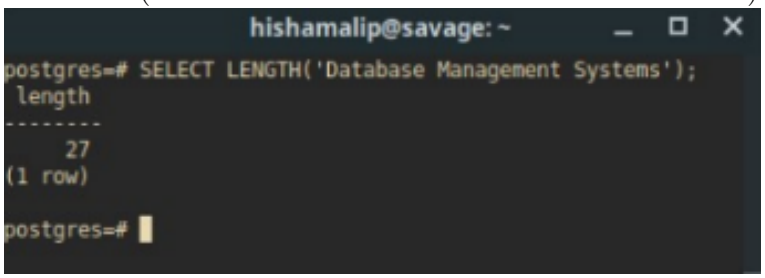


6. Perform INITCAP function on ‘mARKcALAwaY’

INITCAP(’MARKCALAWAY’)

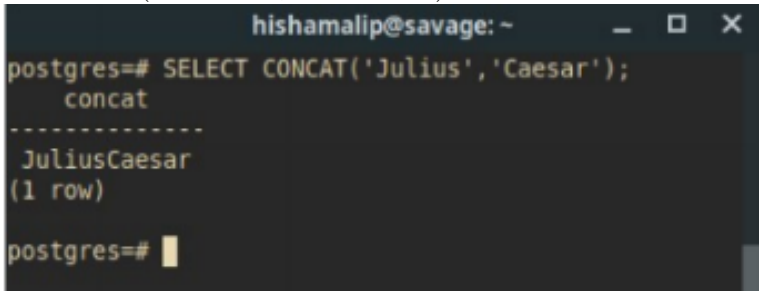
7. Find the length of the string ‘Database Management Systems’.

LENGTH(’DATABASE MANAGEMENT SYSTEMS’)



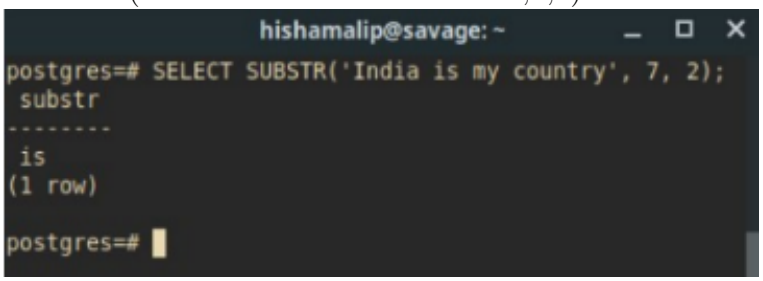
8. Concatenate the strings ‘Julius’ and ‘Caesar’

CONCAT(’JULIUS’,’CAESAR’)



9.Use SUBSTR function to retrieve the substring ‘is’ from the string ‘Indiaismycountry’.

SUBSTR(’INDIA IS MY COUNTRY’,7,2)



RESULT:

Implemented the program for string fucntions and pattern matching using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

8 Experiment 7
JOIN STATEMENTS, SET OPERATIONS, NESTED QUERIES AND GROUPING

AIM:

To get introduced to

- - UNION
- - INTERSECTION
- - MINUS
- - JOIN
- - NESTED QUERIES
- - GROUP BY and HAVING

THEORY:

Joins

A join is a query that combines rows from two or more tables, views, or materialized views ("snapshots"). Oracle performs a join whenever multiple tables appear in the query's FROM clause. The query's select list can select any columns from any of these tables. If any two of these tables have a column name in common, you must qualify all references to these columns throughout the query with table names to avoid ambiguity.

Join Conditions

Most join queries contain WHERE clause conditions that compare two columns, each from a different table. Such a condition is called a join condition.

Equijoins

An equijoin is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns.

Self Joins

A self join is a join of a table to itself.

Cartesian Products

If two tables in a join query have no join condition, Oracle returns their Cartesian product.

Outer Joins

An outer join extends the result of a simple join. An outer join returns all rows that satisfy the join condition and those rows from one table for which no rows from the other satisfy the join condition. Such rows are not returned by a simple join.

Set Operators:

Set operators combine the results of two component queries into a single result. Queries containing set operators are called compound queries. Table lists SQL set operators.

Operator Returns

UNION All - rows selected by either query.

UNION ALL All - rows selected by either query, including all duplicates.

INTERSECT All - distinct rows selected by both queries.

MINUS All - distinct rows selected by the first query but not the second.

All set operators have equal precedence.

NESTED QUERIES

Subquery:

If a sql statement contains another sql statement then the sql statement which is inside another sql statement is called Subquery. It is also known as nested query. The Sql Statement which contains the other sql statement is called Parent Statement.

Nested Subquery:

If a Subquery contains another subquery, then the subquery inside another subquery is called nested subquery.

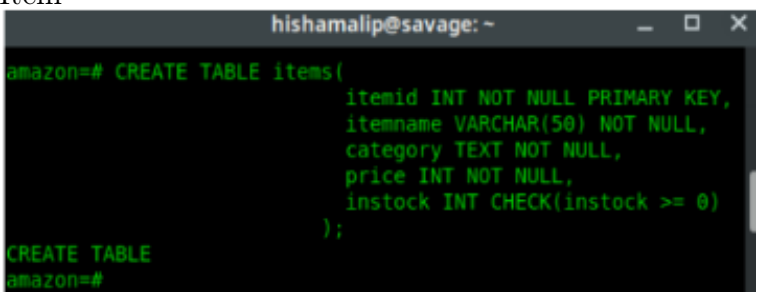
Correlated Subquery:

If the outcome of a subquery is depends on the value of a column of its parent query table then the Sub query is called Correlated Subquery.

QUESTIONS:

Create Items,Orders,Customers,Delivery tables and populate them with appropriate data.

Item



```
hishamalip@savage: ~
amazon=# INSERT INTO items VALUES(100, 'realme 3', 'mobile', 8999, 75);
INSERT 0 1
amazon=# INSERT INTO items VALUES(101, 'sony hd tv', 'tv', 75999, 15);
INSERT 0 1
amazon=# INSERT INTO items VALUES(102, 'one plus 7', 'mobile', 35000, 20),
                                 (103, 'the alchemist', 'book', 250, 100),
                                 (104, 'allen solly', 'shirt', 1139, 35);
INSERT 0 3
amazon=# INSERT INTO items VALUES(105, 'samsung m10', 'mobile', 9999, 100);
INSERT 0 1
amazon=# SELECT * FROM items;
 itemid |   itemname    | category | price | instock
--------+---------------+----------+-------+---------
    100 | realme 3      | mobile   |  8999 |      75
    101 | sony hd tv    | tv       | 75999 |      15
    102 | one plus 7    | mobile   | 35000 |      20
    103 | the alchemist | book     |   250 |     100
    104 | allen solly   | shirt    |  1139 |      35
    105 | samsung m10   | mobile   |  9999 |     100
(6 rows)

amazon=#
```

Orders

```
hishamalip@savage: ~
amazon=# CREATE TABLE customers(
                                custid INT NOT NULL PRIMARY KEY,
                                custname VARCHAR(20) NOT NULL,
                                address VARCHAR(50),
                                state VARCHAR(20) NOT NULL
                               );
CREATE TABLE
amazon=#
```

```
hishamalip@savage: ~
amazon=# INSERT INTO customers VALUES(1000, 'hisham', 'calicut', 'kerala'),
                                     (1001, 'sharuk', 'new delhi', 'delhi'),
                                     (1002, 'vijay', 'chennai', 'tamilnadu'),
                                     (1003, 'alia', 'mumbai', 'maharashtra'),
                                     (1004, 'dhoni', 'ranchi', 'jharkhand'),
                                     (1005, 'sourav', 'trivandrum', 'kerala');
INSERT 0 6
amazon=# SELECT * FROM customers ;
 custid | custname |  address   |    state
--------+----------+------------+-------------
   1000 | hisham   | calicut    | kerala
   1001 | sharuk   | new delhi  | delhi
   1002 | vijay    | chennai    | tamilnadu
   1003 | alia     | mumbai     | maharashtra
   1004 | dhoni    | ranchi     | jharkhand
   1005 | sourav   | trivandrum | kerala
(6 rows)

amazon=#
```

Customers

```
hishamalip@savage: ~
amazon=# CREATE TABLE orders(
                        orderid INT NOT NULL PRIMARY KEY,
                        itemid INT NOT NULL REFERENCES items(itemid)
                                                ON UPDATE CASCADE ON DELETE CASCADE,
                        quandity INT NOT NULL,
                        orderdate DATE,
                        custid INT NOT NULL REFERENCES customers(custid)
                                                ON UPDATE CASCADE ON DELETE CASCADE
                       );
CREATE TABLE
amazon=#
```

```
hishamalip@savage: ~
amazon=# INSERT INTO orders VALUES(1, 100, 3, '2019-08-13', 1000),
amazon-#                         (2, 101, 1, '2017-11-20', 1004),
amazon-#                         (3, 104, 2, '2018-10-27', 1001),
amazon-#                         (4, 103, 5, '2019-07-03', 1005);
INSERT 0 4
amazon=# SELECT * FROM orders;
 orderid | itemid | quandity | orderdate  | custid
---------+--------+----------+------------+--------
       1 |    100 |        3 | 2019-08-13 |   1000
       2 |    101 |        1 | 2017-11-20 |   1004
       3 |    104 |        2 | 2018-10-27 |   1001
       4 |    103 |        5 | 2019-07-03 |   1005
(4 rows)

amazon=#
```

Delivary

```
hishamalip@savage: ~
amazon=# CREATE TABLE delivary(
                        delivaryid INT NOT NULL PRIMARY KEY,
                        orderid INT NOT NULL REFERENCES orders(orderid)
                                                ON UPDATE CASCADE ON DELETE CASCADE,
                           custid INT NOT NULL REFERENCES customers(custid)
                                                ON UPDATE CASCADE ON DELETE CASCADE
                        );
CREATE TABLE
amazon=#
```

```
hishamalip@savage: ~
amazon=# INSERT INTO delivary VALUES(50000, 1, 1000)
amazon-#                           ,(50001, 2, 1004),
amazon-#                            (50002, 4, 1005);
INSERT 0 3
amazon=# SELECT * FROM delivary;
 delivaryid | orderid | custid
------------+---------+--------
      50000 |       1 |   1000
      50001 |       2 |   1004
      50002 |       4 |   1005
(3 rows)

amazon=#
```

1. List the details of all customers who have placed an order

```
hishamalip@savage: ~
amazon=# SELECT customers.custid, custname, address, state
         FROM customers, orders
         WHERE customers.custid = orders.custid;
 custid | custname |  address   |   state
--------+----------+------------+-----------
   1000 | hisham   | calicut    | kerala
   1004 | dhoni    | ranchi     | jharkhand
   1001 | sharuk   | new delhi  | delhi
   1005 | sourav   | trivandrum | kerala
(4 rows)

amazon=#
```

2. List the details of all customers whose orders have been delivered

```
hishamalip@savage: ~
amazon=# SELECT customers.custid, custname, address, state
         FROM customers, delivary
         WHERE customers.custid = delivary.custid;
 custid | custname |  address   |   state
--------+----------+------------+-----------
   1000 | hisham   | calicut    | kerala
   1004 | dhoni    | ranchi     | jharkhand
   1005 | sourav   | trivandrum | kerala
(3 rows)

amazon=#
```

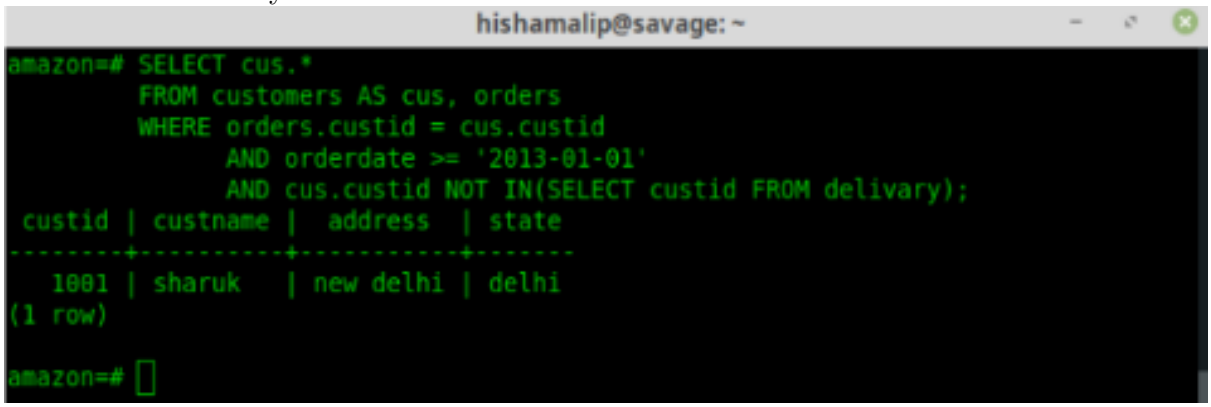
3. Find the orderdate for all customers whose name starts in the letter 'J'

```
hishamalip@savage: ~
amazon=# SELECT orderdate
amazon-# FROM orders, customers
amazon-# WHERE customers.custname LIKE 's%'
amazon-#   AND orders.custid = customers.custid;
 orderdate
------------
 2018-10-27
 2019-07-03
(2 rows)

amazon=#
```

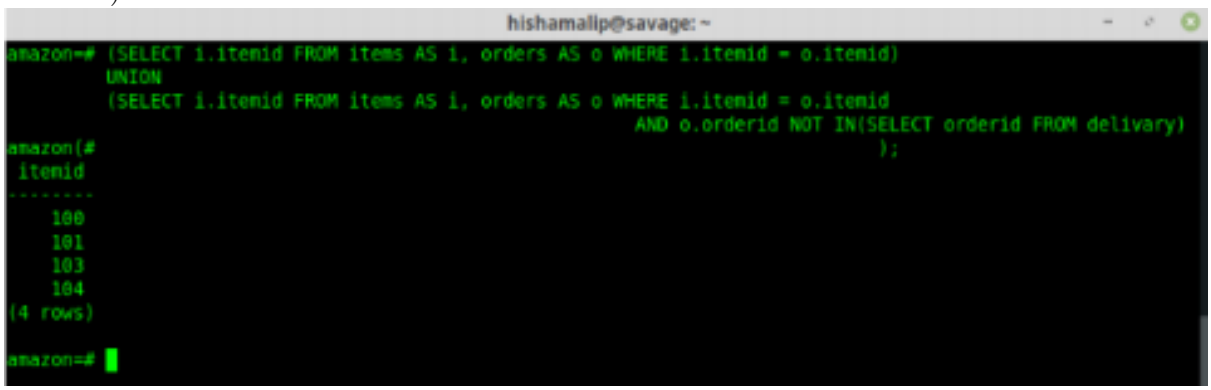
4.Display the name and price of all items bought by the customer 'Mickey'

```
hishamalip@savage: ~

amazon=# SELECT itemname, price
amazon-# FROM items, orders, customers
amazon-# WHERE items.itemid = orders.itemid
amazon-#   AND customers.custid = orders.custid
amazon-#   AND customers.custname = 'hisham';
 itemname | price
----------+-------
 realme 3 |  8999
(1 row)

amazon=#
```

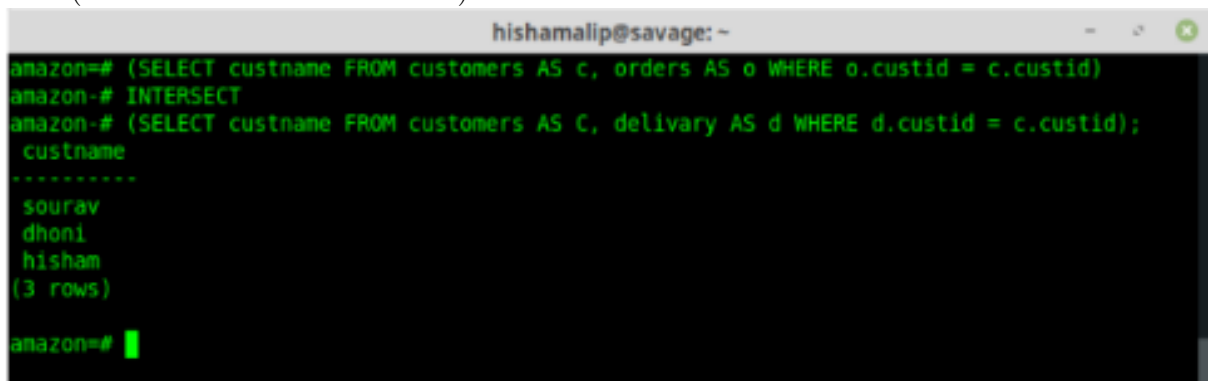
5. List the details of all customers who have placed an order after January 2013 and not received delivery of items.



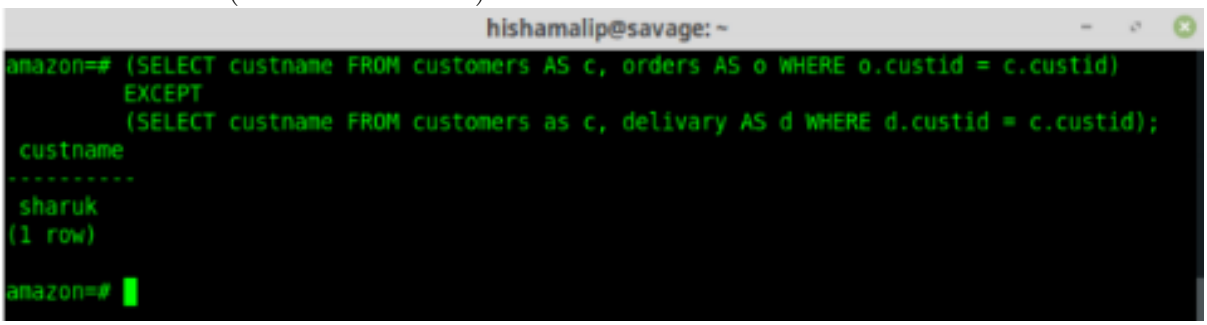
6.Find the itemid of items which has either been ordered or not delivered. (Use SET UNION)



7.Find the name of all customers who have placed an order and have their orders delivered.(Use SET INTERSECTION)

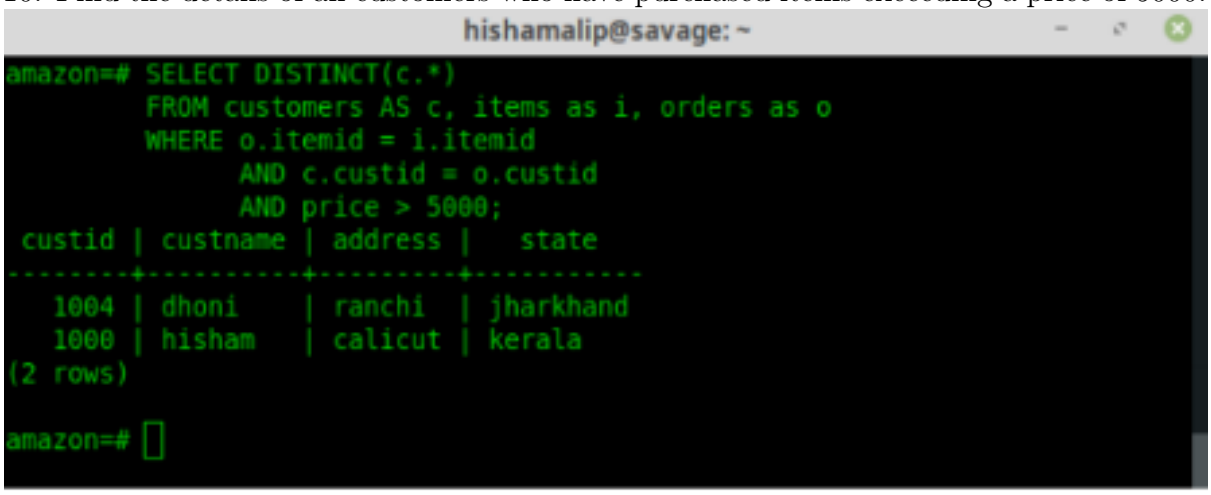


8.Find the custname of all customers who have placed an order but not having their ordersdelivered. (Use SET MINUS).



9. Find the name of the customer who has placed the most number of orders.

10. Find the details of all customers who have purchased items exceeding a price of 5000.



11.Find the name and address of customers who has not ordered a ‘Samsung Galaxy S4’

```
hishamalip@savage: ~
amazon=# (SELECT custname, address FROM customers)
         EXCEPT
         (SELECT c.custname, c.address
          FROM customers AS c, orders AS o, items AS i
          WHERE o.itemid = i.itemid
                AND c.custid = o.custid
                AND itemname = 'sony hd tv');
 custname |  address
----------+------------
 sharuk   | new delhi
 hisham   | calicut
 vijay    | chennai
 sourav   | trivandrum
 alia     | mumbai
(5 rows)

amazon=#
```

12. Perform Left Outer Join and Right Outer Join on Customers and Orders Table.

```
hishamalip@savage: ~
amazon=# SELECT * FROM customers
amazon-# LEFT OUTER JOIN orders
amazon-# ON customers.custid = orders.custid;
 custid | custname |  address   |    state    | orderid | itemid | quandity | orderdate  | custid
--------+----------+------------+-------------+---------+--------+----------+------------+--------
   1000 | hisham   | calicut    | kerala      |       1 |    100 |        3 | 2019-08-13 |   1000
   1004 | dhoni    | ranchi     | jharkhand   |       2 |    101 |        1 | 2017-11-20 |   1004
   1001 | sharuk   | new delhi  | delhi       |       3 |    104 |        2 | 2018-10-27 |   1001
   1005 | sourav   | trivandrum | kerala      |       4 |    103 |        5 | 2019-07-03 |   1005
   1003 | alia     | mumbai     | maharashtra |         |        |          |            |
   1002 | vijay    | chennai    | tamilnadu   |         |        |          |            |
(6 rows)

amazon=#
```

13. Find the details of all customers grouped by state.

```
hishamalip@savage: ~
amazon=# SELECT COUNT(*), state FROM customers GROUP BY state;
 count |    state
-------+-------------
     1 | jharkhand
     1 | delhi
     1 | maharashtra
     2 | kerala
     1 | tamilnadu
(5 rows)

amazon=#
```

14.Display the details of all items grouped by category and having a price greater than the average price of all items.

```
hishamalip@savage: ~

amazon=# SELECT * FROM items
amazon-# GROUP BY category, itemid
amazon-# HAVING price >= (SELECT AVG(price) FROM items);
 itemid |  itemname   | category | price | instock
--------+------------+----------+-------+---------
    101 | sony hd tv | tv       | 75999 |      15
    102 | one plus 7 | mobile   | 35000 |      20
(2 rows)

amazon=#
```

RESULT:

Implemented the program for join, set, nested queries and group by clauses using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

9 Experiment 8
PL/SQL AND SEQUENCE

AIM:

To understand the usage of PL/SQL statements and sequences

THEORY:

PL/SQL is a block-structured language. That is, the basic units (procedures,functions, and anonymous blocks) that make up a PL/SQL program are logical blocks, which can contain any number of nested sub-blocks. Typically, each logical block corresponds to a problem or subproblem to be solved. Thus, PL/SQL supports the divide-and-conquer approach to problem solving called stepwise refinement.

Sequence: The sequence generator generates sequential numbers. Sequence number generation is useful to generate unique primary keys for your data automatically, and to coordinate keys across multiple rows or tables.

QUESTIONS:

1. To print the first 'n' prime numbers.

```
postgres=# CREATE OR REPLACE FUNCTION prime(n integer) RETURNS INTEGER AS $$
postgres$# DECLARE
postgres$#     i INTEGER;
postgres$#     j INTEGER;
postgres$#     flag INTEGER;
postgres$#     count INTEGER;
postgres$# BEGIN
postgres$#     IF n >= 1 THEN
postgres$#         RAISE NOTICE 'first % prime numbers are :', n;
postgres$#     END IF;
postgres$#     count := 1;
postgres$#     i := 2;
postgres$#     WHILE count <= n LOOP
postgres$#         flag = 1;
postgres$#
postgres$#         FOR j IN 2..i-1 LOOP
postgres$#             IF mod(i, j) = 0 THEN
postgres$#                 flag = 0;
postgres$#                 EXIT;
postgres$#             END IF;
postgres$#         END LOOP;
postgres$#
postgres$#         IF flag = 1 THEN
postgres$#             RAISE NOTICE '%', i;
postgres$#             count := count + 1;
postgres$#         END IF;
postgres$#         i := i + 1;
postgres$#     END LOOP;
postgres$# END;
postgres$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# SELECT prime(7);
NOTICE:  first 7 prime numbers are :
NOTICE:  2
NOTICE:  3
NOTICE:  5
NOTICE:  7
NOTICE:  11
NOTICE:  13
NOTICE:  17
ERROR:  control reached end of function without RETURN
CONTEXT:  PL/pgSQL function prime(integer)
postgres=#
```

2. Display the Fibonacci series upto 'n' terms.

```
postgres=# CREATE OR REPLACE FUNCTION fibonacci(n INTEGER) RETURNS INTEGER AS $$
postgres$#  DECLARE
postgres$#      first INTEGER :=  0;
postgres$#       second INTEGER := 1;
postgres$#       temp INTEGER;
postgres$#       k INTEGER := n;
postgres$#       i INTEGER;
postgres$#  BEGIN
postgres$#   RAISE NOTICE 'First % Fibonacci Series is', n;
postgres$#       RAISE NOTICE '%', first;
postgres$#       RAISE NOTICE '%', second;
postgres$#
postgres$#       FOR i IN 2..k-1
postgres$#       LOOP
postgres$#    temp := first + second;
postgres$#    first := second;
postgres$#    second := temp;
postgres$#            RAISE NOTICE '%', temp;
postgres$#       END LOOP;
postgres$#  END;
postgres$#  $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
postgres=# SELECT fibonacci(7);
NOTICE:  First 7 Fibonacci Series is
NOTICE:  0
NOTICE:  1
NOTICE:  1
NOTICE:  2
NOTICE:  3
NOTICE:  5
NOTICE:  8
ERROR:  control reached end of function without RETURN
CONTEXT:  PL/pgSQL function fibonacci(integer)
postgres=#
```

3. Create a table named student grade with the given attributes:
roll, name ,mark1,mark2,mark3, grade.
Read the roll, name and marks from the user. Calculate the grade of the student and insert a tuple into the table using PL/SQL. (Grade= 'PASS' if AVG >40, Grade='FAIL' otherwise)

```
plsql_sequence=# CREATE TABLE student_grade(roll INT NOT NULL PRIMARY KEY, name VARCHAR(10) NOT NULL,
plsql_sequence(#                            mark1 INT NOT NULL, mark2 INT NOT NULL, mark3 INT NOT NULL,
plsql_sequence(#                            grade VARCHAR(10) );
CREATE TABLE
plsql_sequence=# INSERT INTO student_grade(roll, name, mark1, mark2, mark3)
plsql_sequence-# VALUES (1, 'anu', 50, 45, 48), (2, 'manu', 50, 50, 50), (3, 'vinu', 35, 40, 40);
INSERT 0 3
plsql_sequence=# SELECT * FROM student_grade;
 roll | name | mark1 | mark2 | mark3 | grade
------+------+-------+-------+-------+-------
    1 | anu  |    50 |    45 |    48 |
    2 | manu |    50 |    50 |    50 |
    3 | vinu |    35 |    40 |    40 |
(3 rows)

plsql_sequence=# CREATE OR REPLACE FUNCTION calculate_grade() RETURNS VOID AS $$
plsql_sequence$#     UPDATE student_grade
plsql_sequence$#         SET grade = CASE
plsql_sequence$#                         WHEN (mark1 + mark2 + mark3) / 3 > 40 THEN 'Pass'
plsql_sequence$#                         ELSE 'Fail'
plsql_sequence$#                     end;
plsql_sequence$# $$ LANGUAGE SQL;
CREATE FUNCTION
plsql_sequence=# SELECT calculate_grade();
 calculate_grade
-----------------


(1 row)

plsql_sequence=# SELECT * FROM student_grade;
 roll | name | mark1 | mark2 | mark3 | grade
------+------+-------+-------+-------+-------
    1 | anu  |    50 |    45 |    48 | Pass
    2 | manu |    50 |    50 |    50 | Pass
    3 | vinu |    35 |    40 |    40 | Fail
(3 rows)

plsql_sequence=#
```

4. Create table circlearea (rad,area). For radius 5,10,15,20,25, find the area and insert the corresponding values into the table by using loop structure in PL/SQL.

```
plsql_sequence=# CREATE TABLE circle_area(radius INT, area FLOAT);
CREATE TABLE
plsql_sequence=# /* Read step size and limit from user as function parameter */
plsql_sequence-# CREATE OR REPLACE FUNCTION area_calculation(step integer, lim integer) RETURNS VOID AS $$
plsql_sequence$# DECLARE
plsql_sequence$#     area INTEGER;
plsql_sequence$#     temp INTEGER := step;
plsql_sequence$# BEGIN
plsql_sequence$#     TRUNCATE TABLE circle_area;                        -- truncating the table
plsql_sequence$#     LOOP
plsql_sequence$#         area := CEIL(3.14 * temp * temp);              -- area calculation
plsql_sequence$#         INSERT INTO circle_area VALUES(temp, area);    -- add corresponding values into table
plsql_sequence$#         temp := temp + step;                           -- increasing next radius with step
plsql_sequence$#         lim := lim - 1;                                -- decrasing the limit
plsql_sequence$#         EXIT WHEN 0 >= lim;                            -- exit if limit reached
plsql_sequence$#     END LOOP;
plsql_sequence$# END
plsql_sequence$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
plsql_sequence=# SELECT area_calculation(5, 7);
 area_calculation
------------------


(1 row)

plsql_sequence=# SELECT * FROM circle_area;
 radius | area
--------+------
      5 |   79
     10 |  314
     15 |  707
     20 | 1256
     25 | 1963
     30 | 2826
     35 | 3847
(7 rows)

plsql_sequence=#
```

5. Use an array to store the names, marks of 10 students in a class.

Using Loop structures in PL/SQL insert the ten tuples to a table named stud

```
plsql_sequence=# CREATE TABLE student(name TEXT, mark INT);
CREATE TABLE
plsql_sequence=# CREATE OR REPLACE FUNCTION array_input() RETURNS VOID AS $$
plsql_sequence$#     DECLARE
plsql_sequence$#         i INTEGER;
plsql_sequence$#         name array TEXT[] := '{"ARUN", "AMAL", "PETER", "JOSE", "ANNIE", "MARY", "JOSEPH", "MARK", "MIDHUN", "KEVIN"}';
plsql_sequence$#         mark_array INTEGER[] := '{25, 76, 43, 45, 67, 57, 97, 56, 89, 8}';
plsql_sequence$#
plsql_sequence$#     BEGIN
plsql_sequence$#         FOR i IN array_lower(name_array, 1) .. array_upper(name_array, 1) LOOP
plsql_sequence$#             INSERT INTO student(name, mark) VALUES (name_array[i], mark_array[i]);
plsql_sequence$#     END LOOP;
plsql_sequence$#     END
plsql_sequence$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
plsql_sequence=# SELECT array_input();
 array_input
-------------

(1 row)

plsql_sequence=# SELECT * FROM student;
  name  | mark
--------+------
 ARUN   |   25
 AMAL   |   76
 PETER  |   43
 JOSE   |   45
 ANNIE  |   67
 MARY   |   57
 JOSEPH |   97
 MARK   |   56
 MIDHUN |   89
 KEVIN  |    8
(10 rows)

plsql_sequence=#
```

6. Create a sequence using PL/SQL. Use this sequence to generate the primary key values for a table named classcse with attributes roll,name and phone. Insert some tuples using PL/SQL programming.

```
plsql_sequence=# CREATE OR REPLACE FUNCTION my_sequence() RETURNS VOID AS $$
plsql_sequence$# BEGIN
plsql_sequence$#   CREATE TABLE class_cse(roll INT NOT NULL PRIMARY KEY, name VARCHAR(20), phone INT);
plsql_sequence$#   CREATE SEQUENCE roll_no START 1;
plsql_sequence$#   INSERT INTO class_cse VALUES(nextval('roll_no'), 'ARUN', 0482239091),
plsql_sequence$#                              (nextval('roll_no'), 'AMAL', 0484234562),
plsql_sequence$#                              (nextval('roll_no'), 'PETER', 048511234),
plsql_sequence$#                              (nextval('roll_no'), 'JOSE', 048943617),
plsql_sequence$#                              (nextval('roll_no'), 'ANNIE', 048123145);
plsql_sequence$# END
plsql_sequence$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
plsql_sequence=# SELECT my_sequence();
 my_sequence
-------------

(1 row)

plsql_sequence=# SELECT * FROM class_cse;
 roll | name  |   phone
------+-------+-----------
    1 | ARUN  | 482239091
    2 | AMAL  | 484234562
    3 | PETER |  48511234
    4 | JOSE  |  48943617
    5 | ANNIE |  48123145
(5 rows)

plsql_sequence=#
```

RESULT:

Implemented the program for PL/SQL and sequence using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

10 Experiment 9
CURSOR

AIM:

To understand the usage of Cursor.

THEORY:

A PL/SQL construct called a cursor lets you name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return more than one row, you can explicitly declare a cursor to process the rows individually.

QUESTIONS:

1. Create table student (id, name, m1, m2, m3, grade).

Insert 5 tuples into it.

Find the total, calculate grade and update the grade in the table.

```
cursor=# CREATE TABLE student(id INT, name TEXT, m1 INT, m2 INT, m3 INT, grade TEXT);
CREATE TABLE
cursor=# INSERT INTO student(id, name, m1, m2, m3) VALUES(88, 'anu', 39, 67, 92),
                                                        (10, 'jan', 58, 61, 29),
                                                        (30, 'karuna', 87, 79, 77),
                                                        (29, 'jossy', 39, 80, 45),
                                                        (50, 'hisham', 60, 70, 80);
INSERT 0 5
cursor=# SELECT * FROM student ;
 id |  name   | m1 | m2 | m3 | grade
----+---------+----+----+----+-------
 88 | anu     | 39 | 67 | 92 |
 10 | jan     | 58 | 61 | 29 |
 30 | karuna | 87 | 79 | 77 |
 29 | jossy   | 39 | 80 | 45 |
 50 | hisham | 60 | 70 | 80 |
(5 rows)
```

```
cursor=# CREATE OR REPLACE FUNCTION find_grade() RETURNS INTEGER AS $$
cursor$# DECLARE
cursor$#     total FLOAT;
cursor$#     my_cursor CURSOR FOR SELECT * FROM student;
cursor$#     my_record RECORD;
cursor$#
cursor$# BEGIN
cursor$#     OPEN my_cursor;
cursor$#     LOOP
cursor$#         FETCH my_cursor INTO my_record;
cursor$#         EXIT WHEN NOT FOUND;
cursor$#         total = CEIL(my_record.m1 + my_record.m2 + my_record.m3) / 3;
cursor$#
cursor$#         IF  total > 80
cursor$#             THEN UPDATE student SET grade = 'a' WHERE CURRENT OF my_cursor;
cursor$#         ELSIF total > 70 and total <= 80
cursor$#             THEN UPDATE student SET grade = 'b' WHERE CURRENT OF my_cursor;
cursor$#         ELSIF total > 60 and total <= 70
cursor$#             THEN UPDATE student SET grade = 'c' WHERE CURRENT OF my_cursor;
cursor$#         ELSIF total > 40 and total <= 60
cursor$#             THEN UPDATE student SET grade = 'd' WHERE CURRENT OF my_cursor;
cursor$#         ELSE
cursor$#             UPDATE student SET grade = 'f' WHERE CURRENT OF my_cursor;
cursor$#         END IF;
cursor$#
cursor$#     END LOOP;
cursor$#     CLOSE my_cursor;
cursor$#     RETURN 0;
cursor$# END
cursor$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
cursor=# SELECT find_grade();
 find_grade
------------
          0
(1 row)

cursor=# SELECT * FROM student ;
 id | sname  | m1 | m2 | m3 | grade
----+--------+----+----+----+-------
 88 | anu    | 39 | 67 | 92 | c
 10 | jan    | 58 | 61 | 29 | d
 30 | karuna | 87 | 79 | 77 | a
 29 | jossy  | 39 | 80 | 45 | d
 50 | hisham | 60 | 70 | 80 | c
(5 rows)
```

2. Create bankdetails (accno, name, balance, adate).

Calculate the interest of the amount and insert into a new table with fields. (accno, interest). Interest= 0.08*balance.

```
cursor=# CREATE TABLE bank_details(accno INT, name VARCHAR(15), balance INT, adate DATE);
CREATE TABLE
cursor=# insert into bank_details values(1001,'aby',3005,'10-oct-15'),
                                        (1002,'alan',4000,'05-may-95'),
                                        (1003,'amal',5000,'16-mar-92'),
                                        (1004,'jeffin',3500,'01-apr-50'),
                                        (1005,'majo',6600,'01-jan-01');
INSERT 0 5
cursor=# SELECT * FROM bank_details ;
 accno |  name  | balance |   adate
-------+--------+---------+------------
  1001 | aby    |    3005 | 2015-10-10
  1002 | alan   |    4000 | 1995-05-05
  1003 | amal   |    5000 | 1992-03-16
  1004 | jeffin |    3500 | 2050-04-01
  1005 | majo   |    6600 | 2001-01-01
(5 rows)

cursor=#
```

```
cursor=# CREATE TABLE new_bank(accno INT, interest INT);
CREATE TABLE
cursor=# CREATE OR REPLACE FUNCTION calculate_interest() RETURNS VOID AS $$
cursor$# DECLARE
cursor$#     my_cursor CURSOR FOR SELECT * FROM bank_details;
cursor$#     my_record RECORD;
cursor$# BEGIN
cursor$#     OPEN my_cursor;
cursor$#     LOOP
cursor$#         FETCH my_cursor INTO my_record;
cursor$#         EXIT WHEN NOT FOUND;
cursor$#         INSERT INTO new_bank VALUES (my_record.accno, my_record.balance*0.08);
cursor$#     END LOOP;
cursor$#     CLOSE my_cursor;
cursor$# END
cursor$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
cursor=# SELECT calculate_interest();
 calculate_interest
--------------------


(1 row)

cursor=# SELECT * FROM new_bank ;
 accno | interest
-------+----------
  1001 |      240
  1002 |      320
  1003 |      400
  1004 |      280
  1005 |      528
(5 rows)

cursor=#
```

3. Create table peoplelist (id, name, dtjoining, place).If person's experience is above 10 years, put the tuple in table explist (id, name, experience).

```
cursor=# CREATE TABLE people_list(id INT, name VARCHAR(30), dt_joining DATE, place VARCHAR(30));
CREATE TABLE
cursor=# INSERT INTO people_list VALUES (101,'Robert','2005-04-03','CHY'),
cursor-#                                (102, 'Mathew', '2008-06-07', 'CHY'),
cursor-#                                (103, 'Luffy', '2005-04-15', 'FSN'),
cursor-#                                (104, 'Lucci', '2009-08-13', 'KTM'),
cursor-#                                (105, 'Law', '2005-04-12', 'WTC'),
cursor-#                                (106, 'Vivi', '2010-09-21', 'ABA');
INSERT 0 6
cursor=# SELECT * FROM people_list ;
 id  |  name  | dt_joining | place
-----+--------+------------+-------
 101 | Robert | 2005-04-03 | CHY
 102 | Mathew | 2008-06-07 | CHY
 103 | Luffy  | 2005-04-15 | FSN
 104 | Lucci  | 2009-08-13 | KTM
 105 | Law    | 2005-04-12 | WTC
 106 | Vivi   | 2010-09-21 | ABA
(6 rows)

cursor=#
```

```
cursor=# CREATE TABLE experiance_list(id INT, name TEXT, exp INT);
CREATE TABLE
cursor=# CREATE OR REPLACE FUNCTION calculate_experiance() RETURNS INTEGER AS $$
cursor$# DECLARE
cursor$#     my_cursor CURSOR FOR SELECT * FROM people_list;
cursor$#     my_record RECORD;
cursor$#     yd INT;
cursor$# BEGIN
cursor$#     OPEN my_cursor;
cursor$#     LOOP
cursor$#         FETCH FROM my_cursor INTO my_record;
cursor$#         EXIT WHEN NOT FOUND;
cursor$#         yd := date_part('year',age(my_record.dt_joining));
cursor$#         IF yd > 10 THEN
cursor$#             INSERT INTO experiance_list VALUES(my_record.id, my_record.name, yd);
cursor$#         END IF;
cursor$#     END LOOP;
cursor$#     CLOSE my_cursor;
cursor$#     RETURN 0;
cursor$# END
cursor$# $$ LANGUAGE PLPGSQL;

CREATE FUNCTION
cursor=#
cursor=# SELECT calculate_experiance();
 calculate_experiance
----------------------
                    0
(1 row)

cursor=# SELECT * FROM experiance_list ;
 id  |  name  | exp
-----+--------+-----
 101 | Robert |  14
 102 | Mathew |  11
 103 | Luffy  |  14
 105 | Law    |  14
(4 rows)

cursor=#
```

4. Create table employeelist(id,name,monthly salary).

If:

annual salary<60000, increment monthly

salary by 25%.

between 60000 and

200000, increment by 20%

between 200000 and 500000, increment by 15%

annual salary>500000, increment monthly salary by 10%

```
cursor=# create table emp_list(id INT, name varchar(20), m_sal INT);
CREATE TABLE
cursor=# insert into emp_list values(101,'Mathew',55000),
cursor-#                            (102,'Jose',80000),
cursor-#                            (103,'John',250000),
cursor-#                            (104,'Ann',600000);
INSERT 0 4
cursor=# SELECT * FROM emp_list;
 id  |  name  | m_sal
-----+--------+--------
 101 | Mathew |  55000
 102 | Jose   |  80000
 103 | John   | 250000
 104 | Ann    | 600000
(4 rows)

cursor=#
```

```
cursor=# CREATE OR REPLACE FUNCTION update_salary() RETURNS INTEGER AS $$
cursor$# DECLARE
cursor$#     my_cursor CURSOR FOR SELECT * FROM emp_list;
cursor$#     my_record RECORD;
cursor$# BEGIN
cursor$#     OPEN my_cursor;
cursor$#     LOOP
cursor$#         FETCH FROM my_cursor INTO my_record;
cursor$#         EXIT WHEN NOT FOUND;
cursor$#         IF my_record.m_sal*12 < 60000 THEN
cursor$#             UPDATE emp_list SET m_sal = m_sal*1.25  WHERE CURRENT OF my_cursor;
cursor$#         ELSIF my_record.m_sal*12 >= 60000 AND my_record.m_sal*12 < 200000 THEN
cursor$#             UPDATE emp_list SET m_sal = m_sal*1.20  WHERE CURRENT OF my_cursor;
cursor$#         ELSIF my_record.m_sal*12 >= 200000 AND my_record.m_sal*12 < 500000 THEN
cursor$#             UPDATE emp_list SET m_sal = m_sal*1.15  WHERE CURRENT OF my_cursor;
cursor$#         ELSIF my_record.m_sal*12 >= 500000 THEN
cursor$#             UPDATE emp_list SET m_sal = m_sal*1.10  WHERE CURRENT OF my_cursor;
cursor$#         END IF;
cursor$#     END LOOP;
cursor$#     RETURN 0;
cursor$# END;
cursor$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
cursor=#
cursor=# SELECT update_salary();
 update_salary
---------------
             0
(1 row)

cursor=# SELECT * FROM emp_list;
 id  |  name  | m_sal
-----+--------+--------
 101 | Mathew |  60500
 102 | Jose   |  88000
 103 | John   | 275000
 104 | Ann    | 660000
(4 rows)

cursor=#
```

RESULT:

Implemented the program for cursor using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

11 Experiment 10
TRIGGER AND EXCEPTION HANDLING

AIM:

To understand the usage of triggers and exception handling

THEORY:

Triggers:

Triggers are procedures that are stored in the database and implicitly run, or fired, when something happens.

Exception:

It is used to handle run time errors in program

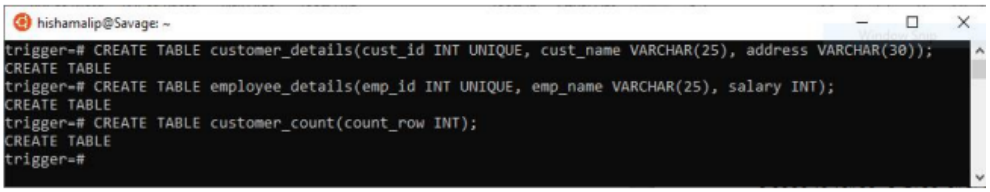
QUESTIONS:

Write PL/SQL programs for the following:

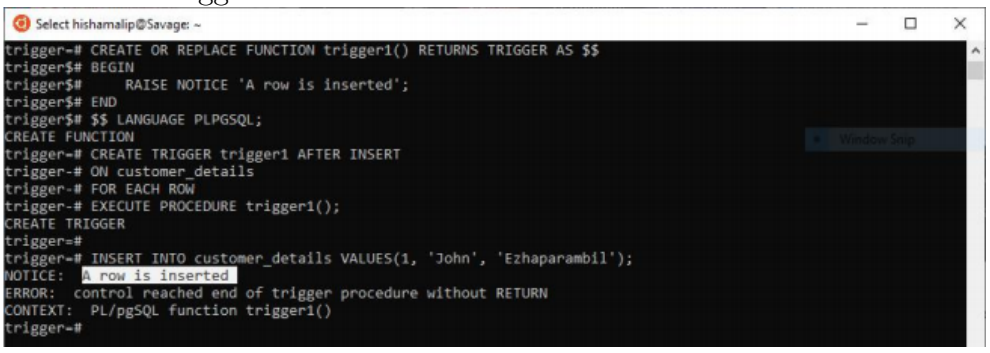
Create a table customerdetails (custid (unique),custname, address).

Create a table empdetails(empid(unique), empname,salary)

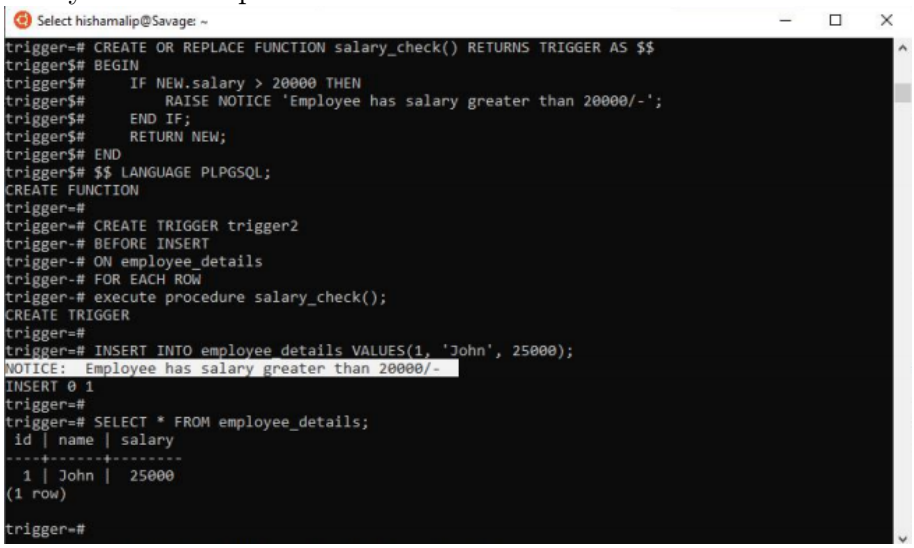
Create table custcount(countrow)



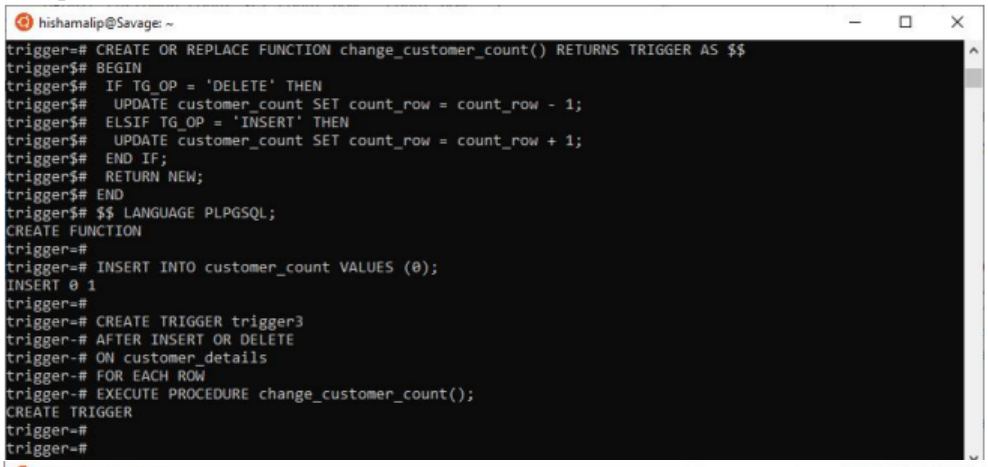
1. Create a trigger whenever a new record is inserted in the customerdetails table.

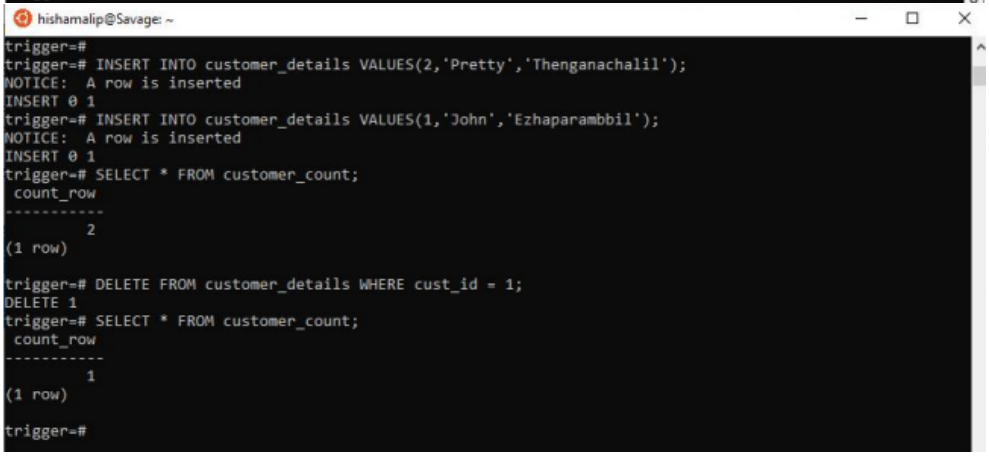


2. Create a trigger to display a message when a user enters a value >20000 in the salary field of empdetails table.



3. Create a trigger w.r.t customerdetails table.Increment the value of countrow (in custcount table) whenever a new tuple is inserted and decrement the value of countrow when a tuple is deleted. Initial value of the countrow is set to 0





4. Create a trigger to insert the deleted rows from empdetails to another table and updated rows to another table.

(Create the tables deleted and updated)

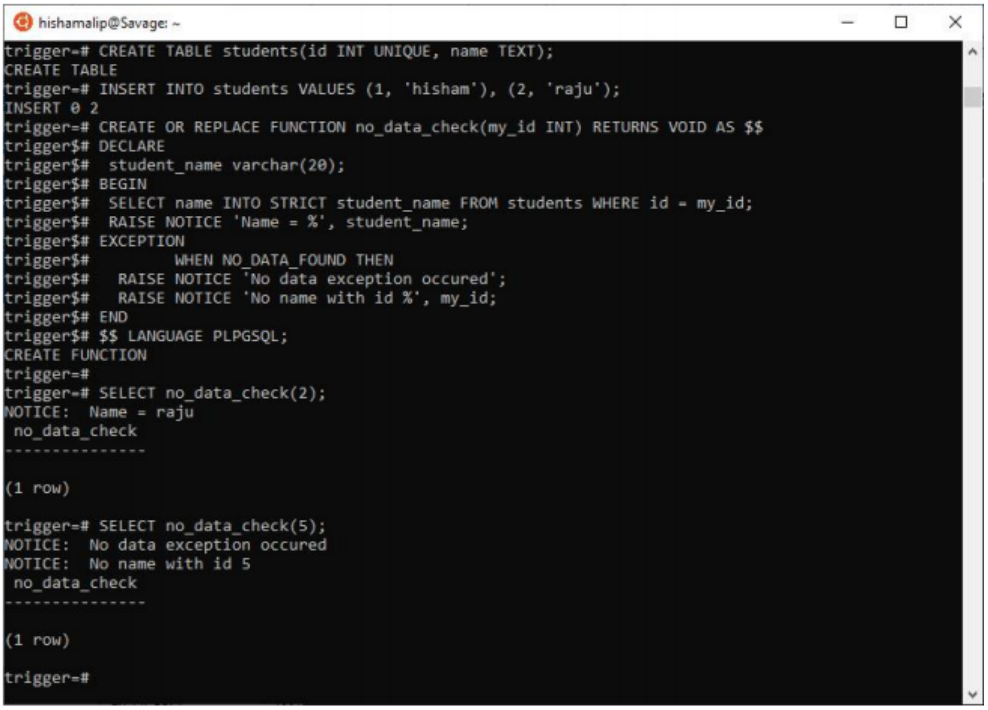
```
hishamalip@Savage: ~
postgres=# CREATE TABLE updated_employee(uemp_id INT, uemp_name TEXT, usalary INT);
CREATE TABLE
postgres=# CREATE TABLE deleted_employee(demp_id INT, demp_name TEXT, dsalary INT);
CREATE TABLE
postgres=#
```

```
hishamalip@Savage: ~
trigger=#
trigger=# CREATE OR REPLACE FUNCTION update_and_delete() RETURNS TRIGGER AS $$
trigger$# BEGIN
trigger$#     IF TG_OP = 'UPDATE' THEN
trigger$#         INSERT INTO updated_employee
trigger$#         VALUES(new.emp_id, new.emp_name, new.salary);
trigger$#     ELSIF TG_OP = 'DELETE' THEN
trigger$#         INSERT INTO deleted_employee
trigger$#         VALUES(old.emp_id, old.emp_name, old.salary);
trigger$#     END IF;
trigger$#     RETURN OLD;
trigger$# END
trigger$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
trigger=#
trigger=# CREATE TRIGGER trigger4
trigger-# AFTER UPDATE OR DELETE
trigger-# ON employee_details
trigger-# FOR EACH ROW
trigger-# EXECUTE PROCEDURE update_and_delete();
CREATE TRIGGER
trigger=#
trigger=# UPDATE employee_details SET salary = salary + 20000 where emp_id = 1;
UPDATE 1
trigger=# SELECT * FROM updated_employee ;
 uemp_id | uemp_name | usalary
---------+-----------+---------
       1 | John      |   45000
(1 row)

trigger=# DELETE FROM employee_details WHERE emp_id = 2;
DELETE 1
trigger=# SELECT * FROM deleted_employee ;
 demp_id | demp_name | dsalary
---------+-----------+---------
       2 | hisham    |   30000
(1 row)

trigger=#
```

5. Write a PL/pgSQL to show divide by zero exception.

```
Select hishamalip@Savage: ~
trigger-# CREATE OR REPLACE FUNCTION division_exception(a FLOAT, b FLOAT) RETURNS FLOAT AS $$
trigger$# BEGIN
trigger$#  RETURN a/b;
trigger$# EXCEPTION
trigger$#  WHEN DIVISION_BY_ZERO THEN
trigger$#    RAISE NOTICE 'Cant divide by zero. Enter another divisor';
trigger$#    RETURN null;
trigger$# END
trigger$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
trigger-# SELECT division_exception(9, 2);
 division_exception
--------------------
                4.5
(1 row)

trigger-# SELECT division_exception(9, 0);
NOTICE:  Cant divide by zero. Enter another divisor
 division_exception
--------------------

(1 row)

trigger-#
```

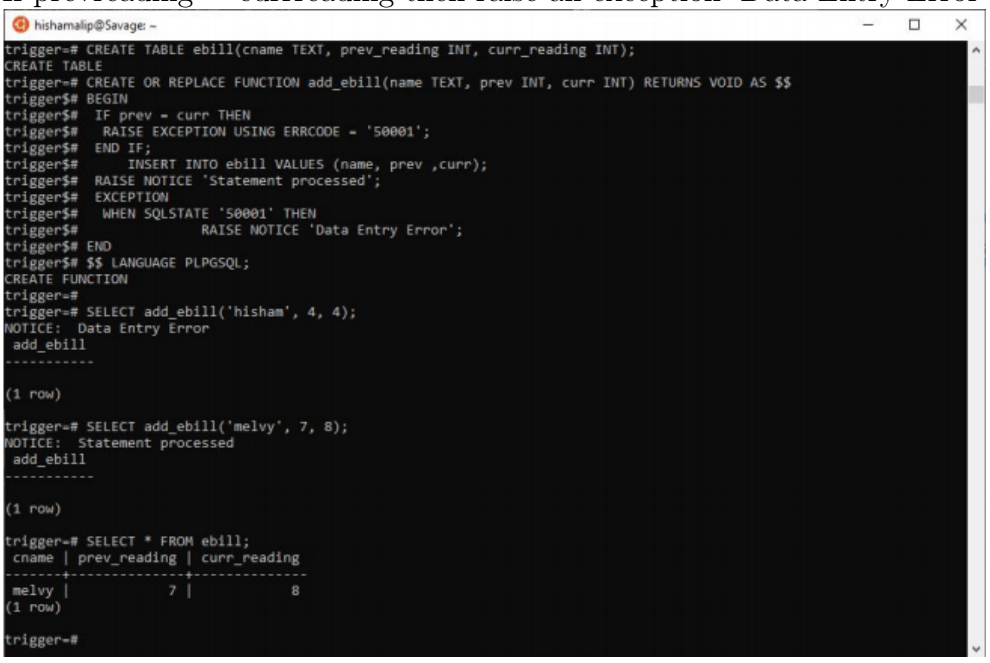
6. Write a PL/pgSQL to show no data found exception.



```
hishamalip@Savage: ~
trigger=# CREATE TABLE students(id INT UNIQUE, name TEXT);
CREATE TABLE
trigger=# INSERT INTO students VALUES (1, 'hisham'), (2, 'raju');
INSERT 0 2
trigger=# CREATE OR REPLACE FUNCTION no_data_check(my_id INT) RETURNS VOID AS $$
trigger$# DECLARE
trigger$#  student_name varchar(20);
trigger$# BEGIN
trigger$#  SELECT name INTO STRICT student_name FROM students WHERE id = my_id;
trigger$#  RAISE NOTICE 'Name = %', student_name;
trigger$# EXCEPTION
trigger$#         WHEN NO_DATA_FOUND THEN
trigger$#   RAISE NOTICE 'No data exception occured';
trigger$#   RAISE NOTICE 'No name with id %', my_id;
trigger$# END
trigger$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
trigger=#
trigger=# SELECT no_data_check(2);
NOTICE:  Name = raju
 no_data_check
---------------

(1 row)

trigger=# SELECT no_data_check(5);
NOTICE:  No data exception occured
NOTICE:  No name with id 5
 no_data_check
---------------

(1 row)

trigger=#
```

7. Create a table with ebill(cname,prevreading,currreading).

If prevreading = currreading then raise an exception 'Data Entry Error'.



```
hishamalip@Savage: ~
trigger=# CREATE TABLE ebill(cname TEXT, prev_reading INT, curr_reading INT);
CREATE TABLE
trigger=# CREATE OR REPLACE FUNCTION add_ebill(name TEXT, prev INT, curr INT) RETURNS VOID AS $$
trigger$# BEGIN
trigger$#  IF prev = curr THEN
trigger$#   RAISE EXCEPTION USING ERRCODE = '50001';
trigger$#  END IF;
trigger$#      INSERT INTO ebill VALUES (name, prev ,curr);
trigger$#  RAISE NOTICE 'Statement processed';
trigger$#  EXCEPTION
trigger$#   WHEN SQLSTATE '50001' THEN
trigger$#               RAISE NOTICE 'Data Entry Error';
trigger$# END
trigger$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
trigger=#
trigger=# SELECT add_ebill('hisham', 4, 4);
NOTICE:  Data Entry Error
 add_ebill
-----------

(1 row)

trigger=# SELECT add_ebill('melvy', 7, 8);
NOTICE:  Statement processed
 add_ebill
-----------

(1 row)

trigger=# SELECT * FROM ebill;
 cname | prev_reading | curr_reading
-------+--------------+--------------
 melvy |            7 |            8
(1 row)

trigger=#
```

RESULT:

Implemented the program for trigger and exception handling using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.

12 Experiment 11
PROCEDURES, FUNCTIONS AND PACKAGES

AIM:

To understand the usage of procedures, functions and packages.

THEORY:

Functions:

A function is a subprogram that computes a value. Functions and procedures are structured alike, except that functions have a RETURN clause.

Procedures:

A procedure is a subprogram that performs a specific action. A procedure has two parts: the specification (spec for short) and the body.

The procedure spec begins with the keyword PROCEDURE and ends with the procedure name or a parameter list.

Parameter declarations are optional. Procedures that take no parameters are written without parentheses. The procedure body begins with the keyword IS and ends with the keyword END followed by an optional procedure name.

The procedure body has three parts:

a declarative part, an executable part, and an optional exception-handling part.

The declarative part contains local declarations, which are placed between the keywords IS and BEGIN. The keyword DECLARE, which introduces declarations in an anonymous PL/SQL block, is not used. The executable part contains statements, which are placed between the keywords BEGIN and EXCEPTION (or END). At least one statement must appear in the executable part of a procedure. The NULL statement meets this requirement.

The exception-handling part contains exception handlers, which are placed between the keywords EXCEPTION and END.

Packages:

A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The specification (spec for short) is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the spec.

QUESTIONS:

1. Create a function factorial to find the factorial of a number. Use this function in a PL/SQL Program to display the factorial of a number read from the user.

```
procedures=# CREATE OR REPLACE FUNCTION factorial(n INT) RETURNS INTEGER AS $$
procedures$# DECLARE
procedures$#     fact INTEGER := 1;
procedures$#     temp INTEGER := n;
procedures$# BEGIN
procedures$#     LOOP
procedures$#         EXIT WHEN temp <= 0;
procedures$#         fact := temp * fact;
procedures$#         temp := temp - 1;
procedures$#     END LOOP;
procedures$#     RETURN fact;
procedures$# END
procedures$# $$ LANGUAGE PLPGSQL;
CREATE FUNCTION
procedures=# SELECT factorial(7);
 factorial
-----------
      5040
(1 row)

procedures=#
```

2. Create a table studentdetails(roll int,marksint, phone int). Create a procedure pr1 to update all rows in the database. Boost the marks of all students by 5/

```
procedures=# CREATE TABLE student_details(roll INT, marks INT, phone BIGINT);
CREATE TABLE
procedures=# INSERT INTO student_details VALUES(1, 70, 9496947423),
procedures-#                                   (2, 85, 9495941358),
procedures-#                                   (3, 78, 8281865009);
INSERT 0 3
procedures=# SELECT * FROM student_details;
 roll | marks |   phone
------+-------+------------
    1 |    70 | 9496947423
    2 |    85 | 9495941358
    3 |    78 | 8281865009
(3 rows)

procedures=#
```

```
procedure=# CREATE OR REPLACE PROCEDURE pr1() AS $$
procedure$# BEGIN
procedure$#     UPDATE student_details
procedure$#     SET marks = marks + (marks * 0.05);
procedure$# END
procedure$# $$ LANGUAGE plpgsql;
CREATE PROCEDURE
procedure=#
procedure=# call pr1();
CALL
procedure=# SELECT * FROM student_details ;
 roll | marks |   phone
------+-------+------------
    1 |    74 | 9496947423
    2 |    89 | 9495941358
    3 |    82 | 8281865009
(3 rows)

procedure=#
```

3. Create table student (id, name, m1, m2, m3, total, grade).Create a function f1 to calculate grade. Create a procedure p1 to update the total and grade.

```
procedure=# CREATE OR REPLACE FUNCTION func_calculate_total_and_grade(stud_id INTEGER) RETURNS VOID AS $$
procedure$# BEGIN
procedure$#     UPDATE student
procedure$#     SET total = m1 + m2 + m3
procedure$#     WHERE id = stud_id;
procedure$#
procedure$#     UPDATE student
procedure$#     SET grade = CASE
procedure$#                     WHEN ((m1 + m2 + m3) / 3) > 40 THEN 'P'
procedure$#                     ELSE 'F'
procedure$#                 END
procedure$#     WHERE id = stud_id;
procedure$# END
procedure$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
procedure=#
procedure=# CREATE OR REPLACE PROCEDURE pro_update_total_and_grade(sid INT, sname TEXT, mark1 INT, mark2 INT, mark3 INT) AS $$
procedure$# BEGIN
procedure$#     INSERT INTO student VALUES(sid, sname, mark1, mark2, mark3);
procedure$#     COMMIT;
procedure$#     PERFORM func_calculate_total_and_grade(sid);
procedure$# END
procedure$# $$ LANGUAGE plpgsql;
CREATE PROCEDURE
procedure=#
procedure=#
```

```
procedure=# CALL pro_update_total_and_grade (1, 'hisham', 55, 87, 63);
CALL
procedure=# SELECT * FROM student;
 id |  name   | m1 | m2 | m3 | total | grade
----+---------+----+----+----+-------+-------
  1 | hisham | 55 | 87 | 63 |   205 | P
(1 row)

procedure=# CALL pro_update_total_and_grade (2, 'john', 35, 58, 21);
CALL
procedure=# CALL pro_update_total_and_grade (3, 'ravi', 87, 36, 94);
CALL
procedure=# SELECT * FROM student;
 id |  name   | m1 | m2 | m3 | total | grade
----+---------+----+----+----+-------+-------
  1 | hisham | 55 | 87 | 63 |   205 | P
  2 | john   | 35 | 58 | 21 |   114 | F
  3 | ravi   | 87 | 36 | 94 |   217 | P
(3 rows)

procedure=#
```

4.Create a package pk1 consisting of the following functions and procedures

- Procedure proc1 to find the sum, average and product of two numbers
- Procedure proc2 to find the square root of a number
- Function named fn11 to check whether a number is even or not
- A function named fn22 to find the sum of 3 numbers

Use this package in a PL/SQL program. Call the functions f11, f22 and procedures pro1, pro2 within the program and display their results.

Creating schema pk1 procedures

```
postgres=# CREATE SCHEMA pk1;
CREATE SCHEMA
postgres=#
```

Creating procedures

```
postgres=# CREATE OR REPLACE PROCEDURE pk1.proc1(num1 NUMERIC, num2 NUMERIC) AS $$
postgres$# DECLARE
postgres$#     sum NUMERIC;
postgres$#     avg FLOAT;
postgres$#     prod NUMERIC;
postgres$# BEGIN
postgres$#     sum := num1 + num2;
postgres$#     avg := (num1 + num2) / 2;
postgres$#     prod := num1 * num2;
postgres$#     RAISE NOTICE ' ';
postgres$#     RAISE NOTICE 'Numbers are % and %', num1, num2;
postgres$#     RAISE NOTICE 'Sum : % ---- Average : % ---- Product : %', sum, avg, prod;
postgres$#     RAISE NOTICE ' ';
postgres$# END
postgres$# $$ LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=#
postgres=#
postgres=# CREATE OR REPLACE PROCEDURE pk1.proc2(num1 FLOAT) AS $$
postgres$# BEGIN
postgres$#     RAISE NOTICE 'Square root of % is %', num1, SQRT(num1);
postgres$#     RAISE NOTICE ' ';
postgres$# END
postgres$# $$ LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=#
```

Creating functions

```
postgres=# CREATE OR REPLACE FUNCTION pk1.fn1(num1 INT) RETURNS VOID AS $$
postgres$# BEGIN
postgres$#     IF num1 % 2 = 0 THEN
postgres$#         RAISE NOTICE '% is even number', num1;
postgres$#     ELSE
postgres$#         RAISE NOTICE '% is odd number', num1;
postgres$#     END IF;
postgres$#     RAISE NOTICE ' ';
postgres$# END
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
postgres=# CREATE OR REPLACE FUNCTION pk1.fn2(num1 INT, num2 INT, num3 INT) RETURNS VOID AS $$
postgres$# BEGIN
postgres$#         RAISE NOTICE 'Sum of %, % and % is : %', num1, num2, num3, (num1+num2+num3);
postgres$#         RAISE NOTICE ' ';
postgres$# END
postgres$# $$ LANGUAGE plpgsql;
CREATE FUNCTION
postgres=#
```

Creating procedure to call functions and procedures inside schema

```
postgres=# CREATE OR REPLACE PROCEDURE pk1.ALL() AS $$
postgres$# BEGIN
postgres$#     CALL pk1.proc1(10, 5);
postgres$#     CALL pk1.proc2(25);
postgres$#     PERFORM pk1.fn1(12);
postgres$#     PERFORM pk1.fn2(2, 6, 1);
postgres$#     COMMIT;
postgres$# END
postgres$# $$ LANGUAGE plpgsql;
CREATE PROCEDURE
postgres=#
postgres=# CALL pk1.ALL();
NOTICE:
NOTICE:  Numbers are 10 and 5
NOTICE:  Sum : 15 ---- Average : 7.5 ---- Product : 50
NOTICE:
NOTICE:  Square root of 25 is 5
NOTICE:
NOTICE:  12 is even number
NOTICE:
NOTICE:  Sum of 2, 6 and 1 is : 9
NOTICE:
CALL
postgres=#
```

Result:

Implemented program for Procedures, functions and packages using Postgresql version: 11.4 ,Ubuntu 18.04 and following output were obtained.