APPLICATION SOFTWARE DEVELOPMENT

[CS333]

LAB MANUAL

SEMESTER V

Department of Computer Science and Engineering

College of Engineering, Trivandrum

Vision and Mission of the Institution

Vision

National level excellence and international visibility in every facet of engineering research and education.

Mission

I To facilitate quality transformative education in engineering and management.

II To foster innovations in technology and its application for meeting global challenges.

III To pursue and disseminate quality research.

IV To equip, enrich and transform students to be responsible professionals for better service to humanity.

Vision and Mission of the Department

Vision

To be a centre of excellence in education and research in the frontier areas of Computer Science and Engineering.

Mission

I To facilitate quality transformative education in Computer Science and Engineering.

II To promote quality research and innovation in technology for meeting global challenges.

III To transform students to competent professionals to cater to the needs of the society.

Program Outcomes

PO1 - Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2 - Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3 -Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4 - Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5 - Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6 -The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7 - Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8 - Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9 - Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10 - Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11 - Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12 -Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives

Consistent with the stated vision and mission of the institute and the Department, the faculty members of the Department strive to train the students to become competent technologists with integrity and social commitment. Within a short span of time after graduation, the graduates shall:

PEO1 - Achieve professional competency in the field of Computer Science and Engineering.

PEO2 - Acquire domain knowledge to pursue higher education and research.

PEO3 -Become socially responsible engineers with good leadership qualities and effective interpersonal skills.

Program Specific Outcomes

PSO1 - Model computational problems by applying mathematical concepts and design solutions using suitable data structures and algorithmic techniques.

PSO2 - Design and develop solutions by following standard software engineering principles and implement by using suitable programming languages and platforms.

PSO3 -Develop system solutions involving both hardware and software modules Program Outcomes (POs)

Course Outcomes

Course Code: CS333
Course Name: Application Software Development Lab

On completion of the course students will be able to,

CO1 Design and model relational database scheme for real-world scenarios using standard design and modeling approaches and implement it using standard DBMS software.

CO2 Design and implement read-only and update queries on databases.

CO3 Design and implement triggers and cursors.

CO4 Implement of stored procedures and control structures (PL/SQL)

CO5 Use front end tools to access database through db connectivity protocols.

CO6 Design and implement complete database projects with front-end and DB Back-end.

Course Educational Objectives

1. To introduce basic commands and operations on database.
2. To introduce stored programming concepts (PL-SQL) using Cursors and Triggers .
3. To familiarize front end tools of database.

Course Code	Course Name	L-T-P-Credits	Year of Introduction
CS334	**ASD Lab**	**0-0-3-1**	**2016**

Pre-requisite: CS208 Principles of Database Design

Course Objectives

- To introduce basic commands and operations on database.
- To introduce stored programming concepts (PL-SQL) using Cursors and Triggers .
- To familiarize front end tools of database.

List of Exercises/ Experiments: (Exercises/experiments marked with * are mandatory. Total 12 Exercises/experiments are mandatory).

1. Creation of a database using DDL commands and writes DQL queries to retrieve information from the database.
2. Performing DML commands like Insertion, Deletion, Modifying, Altering, and Updating records based on conditions.
3. Creating relationship between the databases. *
4. Creating a database to set various constraints. *
5. Practice of SQL TCL commands like Rollback, Commit, Savepoint.
6. Practice of SQL DCL commands for granting and revoking user privileges.
7. Creation of Views and Assertions *
8. Implementation of Build in functions in RDBMS *
9. Implementation of various aggregate functions in SQL *
10. Implementation of Order By, Group By & Having clause. *
11. Implementation of set operators, nested queries and Join queries *
12. Implementation of various control structures using PL/SQL *
13. Creation of Procedures and Functions *
14. Creation of Packages *.
15. Creation of database Triggers and Cursors *
16. Practice various front-end tools and report generation.
17. Creating Forms and Menus
18. Mini project (Application Development using Oracle/ MySQL using Database connectivity)*
    a.Inventory Control System.
    b.Material Requirement Processing.
    c.Hospital Management System.
    d.Railway Reservation System.
    e.Personal Information System.
    f.Web Based User Identification System.
    g.Timetable Management System.
    h.Hotel Management System.

Expected Outcome

The students will be able to

1.Design and implement a database for a given problem using database design principles.

2.Apply stored programming concepts (PL-SQL) using Cursors and Triggers.

3.Use graphical user interface, Event Handling and Database connectivity to develop and deploy applications and applets.

4.Develop medium-sized project in a team.

Evaluation criteria

For Laboratory/ Practical/ Workshop courses:

- **Practical records/ outputs** 30 marks (Internally by the College)
- **Project** 30 marks (Internally by the College)
- **Regular class Viva** 10 marks (Internally by the College)
- **Final written test/ quiz** 30 marks (Internally by the College)

All the above assessments are mandatory to earn credits. If not, the student has to complete the course/ assessments during his free time in consultation with the faculty members. On completion of these, grades will be assigned. In case the Practical/ Laboratory/ Workshop courses are not completed in the semester, grade I (incomplete) will be awarded against the course and the final grade will be given only after the completion of the course/ assessments.

Contents

1 POSTGRESQL INSTALLATION

Exp No 0

Step 1 Update system and install dependencies

It is recommended to update your current system packages if it is a new server instance.

sudo apt update && sudo apt -y upgrade

install vim and wget if not already installed.

sudo apt install -y wget vim

Step 2: Add PostgreSQL 11 APT repository

Before adding repository content to your Ubuntu 18.04 , We need to import the repository signing key:
wget –quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc — sudo apt-key add -

After importing GPG key, add repository contents to your Ubuntu 18.04 system:

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

Verify repository file contents

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

Step 3: Install PostgreSQL 11 on Ubuntu 18.04

The last installation step is for PostgreSQL 11 packages. Run the following commands to install PostgreSQL 11 on Ubuntu 18.04
sudo apt update
sudo apt -y install postgresql-11

Step 4: Set PostgreSQL admin user's password and do testing

Set a password for the default admin user
sudo su - postgres
psql -c "alter user postgres with password 'your password'"

1.1 Result

Successfully installed postgreSQL 11.5 in Ubuntu 18.04.

2 INTRODUCTION TO SQL

Exp 1

2.1 Aim

To get started with SQL

2.2 Theory

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 theACM. 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 sublanguage. 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

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 could 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 to represent relationships.

Creating Tables To create a table, use the SQL command CREATETABLE.
Syntax:
CREATE TABLE table_name(Field_Name Data_type [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 NOTNULL 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 EMP_TAB 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.

Oracle Built-In Datatypes

A datatype associates a fixed set of properties with the values that can be used in a column of a table or in an argument of a procedure or function. These properties cause Oracle to treat values of one datatype differently from values of another datatype. For example, Oracle can add values of NUMBER datatype, but not values of RAW datatype. Oracle supplies the following built-in datatypes:character data types.

- CHAR
- NCHAR
- VARCHAR2 and VARCHAR

- NVARCHAR2
- CLOB
- NCLOB
- LONG

  1. NUMBER datatype
  2. DATE datatype
  3. Binary datatypes

- BLOB
- BFILE
- RAW
- LONG RAW

1. CHAR and NCHAR data types store fixed-length character strings.
2. VARCHAR2 and NVARCHAR2 data types store variable-length character strings. (The VARCHAR datatype is synonymous with the VARCHAR2 datatype.)
3. CLOB and NCLOB data types store single-byte and multibyte character strings of up to four gigabytes.
4. The LONG datatype stores variable-length character strings containing up to two gigabytes, but with many restrictions.
5. This data type is provided for backward compatibility with existing applications; in general, new applications should use CLOB and NCLOB data types to store large amounts of character data.

When deciding which datatype to use for a column that will store alphanumeric data in a table,
consider the following points of distinction:

Space Usage

To store data more efficiently, use the VARCHAR2 datatype. The CHAR data type blank-pads and stores trailing blanks up to a fixed column length for all column values, while the VARCHAR2 datatype does not blank-pad or store trailing blanks for column values.

Comparison Semantics

Use the CHAR data type when you require ANSI compatibility in comparison semantics (when trailing blanks are not important in string comparisons). Use the VARCHAR2 when trailing blanks are important in string comparisons.

Future Compatibility

The CHAR and VARCHAR2 datatypes are and will always be fully supported. At this time, the VARCHAR datatype automatically corresponds to the VARCHAR2 datatype and is reserved for future use.

CHAR, VARCHAR2, and LONG data is automatically converted from the database character set to the character set defined for the user session by the NLS_LANGUAGE parameter, where these are different.

Using the NUMBER 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. You can store positive and negative numbers of magnitude 1 x 10ˆ-130 to 9.99...x10Î125, as well as zero, in a NUMBER column.
For numeric columns you can specify the column as a floating-point number:

Column_name NUMBER Or, you can specify a precision (total number of digits) and scale (number ofdigits to the right of the decimal point): Column_name 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 column stores 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.

Oracle uses its own internal format to store dates. Date data is stored in fixed-length 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 NLS_DATE_FORMAT 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 TO_DATE function with a format mask. For example:
TO_DATE ('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 20th 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 TO_DATE function with a format mask indicating the time portion, as in:
INSERT INTO Birthdays_tab (bname, bday) VALUES ('ANNIE',TO_DATE('13-NOV-9210:56 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 FIXED_DATE initialization parameter allows you to set SYSDATE to a constant; this can be useful for testing.

2.3 Result

3 BASIC SQL QUERIES – I

Exp 2

3.1 Aim

To study the basic sql queries such as

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

3.2 Theory

The SELECT statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

SELECT column1, column2, ...
FROM table_name; Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

SELECT * FROM table_name;

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways. The first way specifies both the column names and the values to be inserted:

INSERT INTO table_name (column1, column2, column3, ...)VALUES (value1, value2, value3, ...);

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

INSERT INTO table_name VALUES (value1, value2, value3, ...);

The SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;

The SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

DELETE FROM table_name WHERE condition;

3.3 Questions

Create the below table named employee and populate it.

Emp_id	Emp_name	Dept	Salary (in US $)
1	Michael	Production	$2500
2	Joe	Production	$2500
3	Smith	Sales	$2250
4	David	Marketing	$2900
5	Richard	Sales	$1600
6	Jessy	Marketing	$1800
7	Jane	Sales	$2000
8	Janet	Production	$3000
9	Neville	Marketing	$2750
10	Richardson	Sales	$1800

```
create table Employee(Emp_id INT NOT NULL,Emp_name VARCHAR(10) NOT NULL,Dept
VARCHAR(20) NOT NULL,Salary INT ,PRIMARY KEY(Emp_id) );
```

```
insert into Employee values('1' ,'Micheal', 'Production', '2500');
insert into Employee values('2', 'Joe', 'Production' ,'2500');
insert into Employee values('3', 'Smith', 'Sales' ,'2250');
insert into Employee values('4', 'David, 'Marketing' ,'2900');
insert into Employee values('5', 'Richard', 'Sales' ,'1600');
insert into Employee values('6', 'Jessy', 'Marketing' ,'1800');
insert into Employee values('7', 'Jane', 'Sales' ,'2000');
insert into Employee values('8', 'Janet', 'Production' ,'3000');
insert into Employee values('9', 'Neville', 'Marketing' ,'2750');
insert into Employee values('10', 'Richardson', 'Sales' ,'1800');
```

1. Display the details of all the employees.

```
select * from Employee;
```

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

```
select emp_id,emp_name from Employee;
```

Emp_id	Emp_name
1	Michael
2	Joe
3	Smith
4	David
5	Richard
6	Jessy
7	Jane
8	Janet
9	Neville
10	Richardson

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

```
delete from Employee where emp_id=10;
```

Emp_id	Emp_name	Dept	Salary (in US $)
1	Michael	Production	$2500
2	Joe	Production	$2500
3	Smith	Sales	$2250
4	David	Marketing	$2900
5	Richard	Sales	$1600
6	Jessy	Marketing	$1800
7	Jane	Sales	$2000
8	Janet	Production	$3000
9	Neville	Marketing	$2750

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

```
insert into Employee values('10', 'Abhi', 'Production');
```

Emp_id	Emp_name	Dept	Salary (in US $)
1	Michael	Production	$2500
2	Joe	Production	$2500
3	Smith	Sales	$2250
4	David	Marketing	$2900
5	Richard	Sales	$1600
6	Jessy	Marketing	$1800
7	Jane	Sales	$2000
8	Janet	Production	$3000
9	Neville	Marketing	$2750
10	Abhi	Production	

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

```
select * from Employee where Dept='Marketing';
```

Emp_id	Emp_name	Dept	Salary (in US $)
4	David	Marketing	$2900
6	Jessy	Marketing	$1800
9	Neville	Marketing	$2750

6. Add the salary details of the newly added employee.

```
update Employee set salary='1000' where emp_id=10;
```

7. Update the salary of Richard to 1900.

```
update Employee set salary='1900' where emp_name=Richard;
```

8. Find the details of all employees who are working for marketing and has a salary greater than2000$.

```
select * from Employee where Dept='Marketing' and salary >'2000';
```

Emp_id	Emp_name	Dept	Salary (in US $)
4	David	Marketing	$2900
9	Neville	Marketing	$2750

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

```
select emp_name from Employee where Dept='Marketing' or Dept='Sales';
```

Emp_name
Smith
David
Richard
Jessy
Jane
Neville

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

```
select emp_name,Dept from Employee where salary>'2300' and salary<'3000'
;
```

Emp_name	Dept
Michael	Production
Joe	Production
David	Marketing
Neville	Marketing

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

```
update Employee set salary=salary*1.2 ;
```

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

```
select emp_name from Employee where salary<2000 or Dept='Sales';
```

3.4 Result

Implemented the program for basic SQL queries using Postgresql (Ubuntu 11.5-1.pgdg18.04+1) in ubuntu 18.04 and following output were obtained.

4 BASIC SQL QUERIES – II

Exp No 3

4.1 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

4.2 Theory

4.3 Questions

Create a table named car_details and populate the table.

```
create table car_details(ID int not null,Name Varchar(10),company varchar(10),
```

ID	Name	company	country	ApproxPrice(in lakhs)
1	Beat	Chevrolet	USA	4
2	Swift	Maruti	Japan	6
3	Escort	Ford	USA	4.2
4	Sunny	Nissan	Japan	8
5	Beetle	Volkswagen	Germany	21
6	Etios	Toyota	Japan	7.2
7	Sail	Chevrolet	USA	5
8	Aria	Tata	India	7
9	Passat	Volkswagen	Germany	25
10	SX4	Maruti	Japan	6.7

```
country varchar(10),ApproxPrice real);
```

```
insert into car_details values('1','Beat','Chevrolet','USA','4');
```

```
insert into car_details values('2','Swift','Maruti','Japan','6');
insert into car_details values('3','Escort','Ford','USA','4.2');
insert into car_details values('4','Sunny','Nissan','Japan','8');
insert into car_details values('5','Etios','Toyoto','Japan','7.2');
insert into car_details values('6','Beetle','Volkswagen','Germany','21');
insert into car_details values('7','Sail','Chevrolet','USA','5');
insert into car_details values('8','Aria','Tata','India','7');
insert into car_details values('9','Passat','Volkswagen','Germany','25');
insert into car_details values('10','SX4','Maruti','Japan','6.7');
```

0. Display the details of all cars

```
select * from car_details;
```

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

```
select distinct company from car_details;
```

company
Chevrolet
Maruti
Ford
Nissan
Volkswagen
Toyota
Tata

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

```
select distinct country from car_details;
```

country
USA
Japan
Germany
India

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

```
select * from car_details where approxprice>='4' and approxprice<=7;
```

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

```
select name,company from car_details where country='Japan' and approxprice<='6';
```

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

```
select name,company from car_details where company='Nissan' or approxprice>'20';
```

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

```
select name from car_details where company in ('Maruti' , 'Ford');
```

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

```
alter table car_details add year int;
update car_details set year='2015';
```

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

```
select name as Car_name from car_details ;
```

9 Rename the attribute name to car_name.

```
alter table car_details rename name to car_name;
```

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

```
select car_name as cars_toyoto from car_details where company='Toyoto';
```

11 List the details of all cars in alphabetical order.

```
select * from car_details order by car_name;
```

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

```
select * from car_details order by approxprice;
```

4.4 Result

Implemented the program for basic SQL queries using Postgresql (Ubuntu 11.5-1.pgdg18.04+1) in ubuntu 18.04 and following output were obtained.

5 INTRODUCTION TO AGGREGATE FUNCTIONS

Exp No 4

5.1 Aim

Introduction to Aggregate functions

- AVG()
- MIN()
- COUNT()
- MAX()
- SUM()

5.2 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 (All): returns the total number of rows. It is same like count(*)
- **Max(fieldname)** Returns the maximum value of the field
- **Min(fieldname)** Returns the maximum value of the field

5.3 Questions

Create a table named student and populate 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.

Roll No	Name	Physics	Chemistry	Maths
1	Adam	20	20	33
2	Bob	18	9	41
3	Bright	22	7	31
4	Duke	13	21	20
5	Elvin	14	22	23
6	Fletcher	2	10	48
7	Georgina	22	12	22
8	Mary	24	14	31
9	Tom	19	15	24
10	Zack	8	20	36

```
   create table student(rollno int not null,name varchar(10),physics int,chemistry
int , maths int,primary key(rollno));


insert into student values('1','Adam','20','20','33');
insert into student values('2','Bob','18','9','41');
insert into student values('3','Bright','22','7','31');
insert into student values('4','Duke','13','21','20');
insert into student values('5','Elvin','14','22','23');
insert into student values('6','Fetcher','2','10','48');
insert into student values('7','Georgina','22','12','22');
insert into student values('8','Mary','24','14','31');
insert into student values('9','Tom','19','15','24');
insert into student values('10','Zack','8','20','36');
```

0. Display the Table

```
select * from student;
```

1 Find the class average for the subject 'Physics'

```
select avg(physics) from student;
```

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

```
select max(maths) as highest_marks_maths from student;
```

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

```
select min(chemistry) as lowest_marks_chemistry from student;
```

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

```
select count(*) from student where physics>='12';
```

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

```
select * from student where physics>='12' and chemistry>='12' and maths>='25';
```

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

```
alter table student add column total int ;
alter table student add column p_or_f char(1) ;
update student set total=physics+chemistry+maths;
update student set p_or_f='p' where physics>=12 and chemistry>=12 and maths>=25;
update student set p_or_f='f' where physics<12 or chemistry<12 or maths<25;
select * from student order by total desc;
```

7. Find pass percentage of the class for mathematics.

```
select count(rollno)*100/(select count(rollno) from student) as maths_pass_percentag
from student where maths>='25';
```

8 Find the overall pass percentage for all class.

```
select count(rollno)*100/(select count(rollno) from student) as pass_percentage
from student where p_or_f='p';
```

9 Find the class average.

```
select avg(total) from student;
```

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

```
 select count(*) from student where p_or_f='p';
```

5.4 Result

Implemented the program for Aggregate functions in Postgresql 11.5 in ubuntu 18.04 and following output were obtained.

6 DATA CONSTRAINTS AND VIEWS

Exp No 5

6.1 Aim

To study about various data constraints and views in SQL.

6.2 Theory

Using Referential Integrity Constraints

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.

Foreign keys can be comprised of multiple columns. However, a composite foreign key must reference a composite primary or unique key of the exact same structure (the same number of columns and datatypes). Because composite primary and unique keys are limited to 16 columns, a composite foreign key is also limited to 16 columns.

Nulls and Foreign Keys

By default (without any NOT NULL or CHECK clauses), and in accordance with the ANSI/ISO standard, the FOREIGNKEY constraint enforces the "match none" rule for composite foreign keys. The "full" and "partial" rules can also be enforced by using CHECK and NOTNULL constraints, as follows:

- To enforce the "match full" rule for nulls in composite foreign keys, which requires that all components of the key be null or all be non-null, define a CHECK constraint that allows only all nulls or all non-nulls in the composite foreign key as follows, assuming a composite key comprised of columns A, B, and C:

  CHECK ((A IS NULL AND B IS NULL AND C IS NULL) OR (A IS NOT NULL AND B IS NOT NULL AND C IS NOT NULL))

- In general, it is not possible to use declarative referential integrity to enforce the "match partial" rule for nulls in composite foreign keys, which requires the non-null portions of the key to appear in the corresponding portions in the primary or unique key of a single row in the referenced table. You can often use triggers to handle this case.

Relationships Between Parent and Child Tables

Several relationships between parent and child tables can be determined by the other types of integrity constraints defined on the foreign key in the child table.

No Constraints on the Foreign Key

When no other constraints are defined on the foreign key, any number of rows in the child table can reference the same parent key value. This model allows nulls in the foreign key.

This model establishes a "one-to-many" relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. An example of such a relationship is shown in Figure 4-3 on page 8 between EMP_TAB and DEPT_TAB; each department (parent key) has many employees (foreign key), and some employees might not be in a department (nulls in the foreign key).

NOT NULL Constraint on the Foreign Key

When nulls are not allowed in a foreign key, each row in the child table must explicitly reference a value in the parent key because nulls are not allowed in the foreign key. However, any number of rows in the child table can reference the same parent key value.

This model establishes a "one-to-many" relationship between the parent and foreign keys. However, each row in the child table must have a reference to a parent key value; the absence of a value (a null) in the foreign key is not allowed. The same example in the previous section can be used to illustrate such a relationship. However, in this case, employees must have a reference to a specific department.

UNIQUE Constraint on the Foreign Key

When a UNIQUE constraint is defined on the foreign key, one row in the child table can reference a parent key value. This model allows nulls in the foreign key.

This model establishes a "one-to-one" relationship between the parent and foreign keys that allows undetermined values (nulls) in the foreign key. For example, assume that the EMP_TAB table had a column named MEMBERNO, referring to an employee's membership number in the company's insurance plan. Also, a table named INSURANCE has a primary key named MEMBERNO,

and other columns of the table keep respective information relating to an employee's insurance policy. The MEMBERNO in the EMP_TAB table should be both a foreign key and a unique key:

1. To enforce referential integrity rules between the EMP_TAB and INSURANCE tables (the FOREIGN KEY constraint)
2. To guarantee that each employee has a unique membership number (the UNIQUE key constraint)

UNIQUE and NOT NULL Constraints on the Foreign Key

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.

This model establishes a "one-to-one" relationship between the parent and foreign keys that does not allow undetermined values (nulls) in the foreign key. If you expand the previous example by adding a NOTNULL constraint on the MEMBERNO column of the EMP_TAB table, in addition to guaranteeing that each employee has a unique membership number, then you also ensure that no undetermined values (nulls) are allowed in the MEMBERNO column of the EMP_TAB table.

6.3 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

```
    create table Subjects(sub_id int not null,sub_name varchar(10) not
null);

insert into subjects values( '1', "Maths");
insert into subjects values( '2', 'Physics');
insert into subjects values( '3','Chemistry');
insert into subjects values( '4','English');
```

0. Display the Table

```
select * from subjects;
```

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

```
alter table subjects add primary key(sub_id);
```

b. Create a table named Staff with the given attributes

- -staffid (Should be UNIQUE)
- -staffname
- -dept
- -Age (Greater than 22)
- -Salary (Less than 35000)

STAFF_ID	STAFF_NAME	DEPT	AGE	SALARY
1	John	Purchasing	24	30000
2	Sera	Sales	25	20000
3	Jane	Sales	28	25000

```
create table Staff(staff_id int not null unique,staff_name varchar(10),
dept varchar(10),age int ,salary int,check(age>22),check(salary<35000));

insert into staff values('1','John','Purchasing','24','30000');
insert into staff values('2','Sera','Sales','25','20000');
insert into staff values('3','Jane','Sales','28','25000');
```

i)Delete the check constraint imposed on the attribute salary)

```
alter table staff drop constraint staff_salary_check;
```

ii)Delete the unique constraint on the attribute staffid.

```
alter table staff drop constraint staff_staff_id_key;
```

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)

BANKCODE	BANK NAME	HEADOFFICE	BRANCHOFFICE	SALARY
AAA	SIB	Ernakulam	6	30000
BBB	Federal	Kottayam	5	20000
CCC	Canara	Trivandrum	3	25000

```
create table Bank(bankcode varchar(3),bank_name varchar(10),
headoffice varchar(10),branchoffice int );


alter table bank add constraint primarykey primary key (bankcode);
alter table bank add constraint branchoffice_check check(branchoffice>0);
insert into bank values('AAA','SIB','Ernakulam','6');
insert into bank values('BBB','Federal','Kottayam','5');
insert into bank values('CCC','Canara','Trivandrum','3');
```

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)

BRANCH_ID	BRANCHNAME	BANKID
1	Kottayam	CCC
2	New Delhi	AAA

```
   create table Branch(branchid int ,branchname varchar(10) default 'New
Delhi', bankid varchar(3),primary key(branchid) );

alter table Branch add constraint branch_bankid_fkey foreign key(bankid)
references bank(bankcode) on update cascade on delete cascade;

insert into Branch values('1','Kottayam','CCC');
insert into Branch(branchid,bankid) values('2','AAA');
insert into bank values('SBT','Indian','Delhi','7');
insert into Branch values('5','Calicut','SBT');
```

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

```
delete from bank where bankcode='SBT';
```

iv) Drop the Primary Key using ALTER command

```
alter table branch drop constraint branch_pkey;
insert into branch values('1','PPP','CCC');
```

2. Create a View named sales_staff to hold the details of all staff working in sales Department

```
create view sales_staff as select * from staff where dept='Sales';
select * from sales_staff;
```

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

Constraint name	Column	Constraint
Pk	branch_id	Primary key
Df	branch_name	Default :"New Delhi"
Fk	bankid	Foreign key/References

```
    drop table Branch;
create table Branch(branchid int ,branchname varchar(10)
constraint Df default 'New Delhi' ,bankid varchar(3),
constraint pk primary key(branchid),
constraint Fk foreign key(bankid) references bank(bankcode) on update cascade
on delete cascade);
```

i) Delete the default constraint in the table

```
alter table branch alter branchname drop default;
```

ii) Delete the primary key constraint

```
alter table branch drop constraint Pk;
```

4. Update the view sales_staff to include the details of staff belonging to sales department whose salary is greater than 20000.

```
create or replace view sales_staff as(select * from staff
where salary>'20000' and dept='Sales');
```

5. Delete the view sales_staff.

```
drop view sales_staff;
```

6.4 Result

Implemented the programs using Data constrains and views in Postgresql 11.5 (Ubuntu 11.5-1.pgdg18.04+1) in ubuntu 18.04 and following output were obtained.

7 STRING FUNCTIONS & PATTERN MATCHING

Exp No 6

7.1 Aim

Study String Functions & Pattern Matching

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

7.2 Thoery

The main string functions are as follows:

Length()	*Length(fname/string)*	Gives the length of the string.
Lower()	*Lower(fname/string*	Gives the content in lowercase letters
Upper()	*Upper(fname/string)*	Gives the content in upper case letters
Concat()	*Concat (fname/string, fname/string)*	Combines the first and second string .
Lpad()	*Lpad(fname/string,length,character)*	Returns char1, left-padded to length n with the sequence of characters in char2
Rpad()	*Rpad(fname/string,length,character)*	Returns char1, right-padded to length n with char2
Rtrim()	*Rtrim(fname/string,substring)*	Returns char, with all the rightmost characters that appear in set removed
Instr()	*Instr(fname/string,substring,n,m)*	Searches char1 beginning with its nth character for the nth occurrence of char2 and returns the position of the character in char1 that is the first character of this occurrence.
Substr()	*Substr(fname/string,n,m)*	Returns a portion of char, beginning at character m, n characters long.

7.3 Questions

A. Create a table named acct_details and populate the table .

Acct_No	Branch	Name	Phone
A40123401	Chicago	Mike Adams	(378) 400-1234
A40123402	Miami	Diana George	(372) 420-2345
B40123403	Miami	Diaz Elizabeth	(371) 450-3456
B40123404	Atlanta	Jeoffrey George	(370) 460-4567
B40123405	New York	Jennifer Kaitlyn	(373) 470-5678
C40123406	Chicago	Kaitlyn Vincent	(318) 200-3235
C40123407	Miami	Abraham Gottfield	(328) 300-2256
C50123408	New Jersey	Stacy Williams	(338) 400-5237
D50123409	New York	Catherine George	(348) 500-6228
D50123410	Miami	Oliver Scott	(358) 600-7230

```
    create table acc_details(Acct_no char(9) primary key,Branch varchar(10),
name varchar(20),phone int);


insert into acc_details values('A40123401','Chicago' ,'Mike Adams' ,'(378)400-1234')
insert into acc_details values('A40123402','Miami','Diana George','(372)420-2345');
insert into acc_details values('B40123403','Miami','Diaz Elizabeth','(371)450-3456')
insert into acc_details values('B40123404','Atlanta','Jeoffrey George','(370)460-456
insert into acc_details values('B40123405','New York','Jennifer Kaitlyn','(373)470-5
insert into acc_details values('C40123406','Chicago','Kaitlyn Vincent','(318)200-323
insert into acc_details values('C40123407','Miami','Abraham Gottfield','(328)300-225
insert into acc_details values('C50123408','New Jersey','Stacy Williams','(338)400-5
insert into acc_details values('D50123409','New York','Catherine George','(348)500-6
insert into acc_details values('D50123410','Miami','Oliver Scott','(358)600-7230');
```

0. Display the Table

```
select * from acc_details;
```

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

```
select name from acc_details where name like 'D%';
```

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

```
select branch from acc_details where branch like '%New%';
```

3. List all the names in Upper Case Format

```
select upper(name) from acc_details;
```

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

```
select name from acc_details where name like '___n%n';
```

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

```
select name from acc_details where name like 'D_a%' and name like '%Eli%';
```

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

```
select name from acc_details where acct_no like '%6';
```

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

```
update acc_details set name=upper(name);
```

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

```
select name from acc_details where lower(name) like '%t'
```

9. List all the names in reverse

```
 select reverse(name) as reverse_name from acc_details;
```

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

```
select lpad(phone,15,'+1') from acc_details;
```

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

```
select ltrim(acct_no,'ABCD') as acct_no,name,branch,phone from acc_details;
```

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

```
select * from acc_details where upper(name) like '%WILLIAMS%' or acct_no
like '_4%';
```

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

1. Find the reverse of the string ' nmutuAotedOehT'.

```
SELECT REVERSE('NMUTUAOTEDOEHT');
```

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

```
SELECT LTRIM('123231XYZTECH','123XYZ');
```

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

```
SELECT RTRIM('COMPUTER');
```

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

```
SELECT RPAD('COMPUTER',12,'X');
```

5. Use POSITION function to find the first occurrence of 'e' in the string 'Welcome to Kerala'.

```
SELECT POSITION('E' in 'WELCOMETOKERALA');
```

6. Perform INITCAP function on 'mARKcALAwaY'.

```
SELECT INITCAP('MARK CALAWAY');
```

7. Find the length of the string 'Database Management Systems'..

```
SELECT LENGTH('DATABASE MANAGEMENT SYSTEMS');
```

8. Concatenate the strings 'Julius' and 'Caesar'.

```
SELECT CONCAT('JULIUS','CAESAR');
```

9. Use SUBSTR function to retrieve the substring 'is' from the string 'India is my country'.

```
SELECT SUBSTR('INDIA IS MY COUNTRY',7,2);
```

10. Use INSTR function to find the second occurrence of 'k' from the last. The string is 'Making of a King'.

```
SELECT INSTR('MAKING OF A KING','K', -1,2);
```

7.4 Result

Implemented the programs using String Functions in Postgresql 11.5 (Ubuntu 11.5-1.pgdg18.04+1) in ubuntu 18.04 and following output were obtained.

8 JOIN STATEMENTS, SET OPERATIONS, NESTED QUERIES AND GROUPING

Exp No 7

8.1 Aim

To get introduced to

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

8.2 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. To execute a join, Oracle combines pairs of rows, each containing one row from each table, for which the join condition evaluates to TRUE. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on join conditions containing columns of the joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and, in the case of the cost-based optimization approach, statistics for the tables.

In addition to join conditions, the WHERE clause of a join query can also contain other conditions that refer to columns of only one table. These conditions can further

restrict the rows returned by the join query.

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. Depending on the internal algorithm the optimizer chooses to execute the join, the total size of the columns in the equijoin condition in a single table may be limited to the size of a data block minus some overhead. The size of a data block is specified by the initialization parameter DB_BLOCK_SIZE.

Self Joins

A self join is a join of a table to itself. This table appears twice in the FROM clause and is followed by table aliases that qualify column names in the join condition. To perform a self join, Oracle combines and returns rows of the table that satisfy the join condition.

Cartesian Products

If two tables in a join query have no join condition, Oracle returns their Cartesian product. Oracle combines each row of one table with each row of the other. A Cartesian product always generates many rows and is rarely useful. For example, the Cartesian product of two tables, each with 100 rows, has 10,000 rows. Always include a join condition unless you specifically need a Cartesian product. If a query joins three or more tables and you do not specify a join condition for a specific pair, the optimizer may choose a join order that avoids producing an intermediate 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. To write a query that performs an outer join of tables A and B and returns all rows from A, apply the outer join operator (+) to all columns of B in the join condition. For all rows in A that have no matching rows in B, Oracle returns NULL for any select list expressions containing columns of B.

Outer join queries are subject to the following rules and restrictions:

- The (+) operator can appear only in the WHERE clause or, in the context of left-correlation (that is, when specifying the TABLE clause) in the FROM

clause, and can be applied only to a column of a table or view.

- If A and B are joined by multiple join conditions, you must use the (+) operator in all of these conditions. If you do not, Oracle will return only the rows resulting from a simple join, but without a warning or error to advise you that you do not have the results of an outer join.
- The (+) operator can be applied only to a column, not to an arbitrary expression. However, an arbitrary expression can contain a column marked with the (+) operator.
- A condition containing the (+) operator cannot be combined with another condition using the OR logical operator.
- A condition cannot use the IN comparison operator to compare a column marked with the (+) operator with an expression.
- A condition cannot compare any column marked with the (+) operator with a subquery.
- If the WHERE clause contains a condition that compares a column from table B with a constant, the (+) operator must be applied to the column so that Oracle returns the rows from table A for which it has generated NULLs for this column. Otherwise Oracle will return only the results of a simple join.
- In a query that performs outer joins of more than two pairs of tables, a single table can be the NULL-generated table for only one other table. For this reason, you cannot apply the (+) operator to columns of B in the join condition for A and B and the join condition for B and C.

*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. If a SQL statement contains multiple set operators, Oracle evaluates them from the left to right if no parentheses explicitly specify another order.

The corresponding expressions in the select lists of the component queries of a compound query must match in number and datatype. If component queries select character data, the datatype of the return values are determined as follows:

- If both queries select values of datatype CHAR, the returned values have datatype CHAR.
- If either or both of the queries select values of datatype VARCHAR2, the returned values have datatype VARCHAR2.

Set Membership (IN NOT IN):

NOT IN Operator

If any item in the list following a NOT IN operation is null, all rows evaluate to UNKNOWN (and no rows are returned). For example, the following statement returns the string 'TRUE' for each row:

SELECT 'TRUE' FROM emp WHERE deptno NOT IN (5,15);

However, the following statement returns no rows:

SELECT 'TRUE' FROM emp WHERE deptno NOT IN (5,15,null);

The above example returns no rows because the WHERE clause condition evaluates to:

deptno != 5 AND deptno != 15 AND deptno != null Because all conditions that compare a null result in a null, the entire expression results in a null. This behavior can easily be overlooked, especially when the NOT IN operator references a subquery.

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.

8.3 Questions

Amazon is one of the largest online stores operating in the United States of America. They are maintaining four tables in their database. The Items table, Customers table, Orders table and Delivery table. Each of these tables contains the following attributes:

- Items: - itemid (primary key) Itemname(type =varchar(50)) category(type=varchar(10)),Price, Instock (type=int, greater than or equal to zero)
- Customers:- custid (primary key) Custname, Address , state
- Orders:- orderid (primary key) , Itemid(refers to itemid of Items table),Quantity (type=int) Orderdate (type=date),custid(refers to custid of customers table)
- Delivery:- delivery id (primary key),Custid (refers to custid in customers table) Orderid (refers to ordered in orders table)

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

```
 create table items(
itemid int not null,
itemname varchar(50) not null,
category varchar(20) not null,
price int not null,
instock int ,
constraint checkstock check(instock >0),
constraint pkey primary key(itemid));



insert into items values(5,'sony z5 premium','electronics',5005,1);
insert into items values(4,'Samsung Galaxy S4','electronics',5005,1);
insert into items values(3,'One Plus 7','electronics',6006,2);
insert into items values(2,'Iphone X','electronics',7007,6);
insert into items values(1,'Xiomi','electronics',1001,6);
```

0. Display the details items table

```
select * from items;
```

```
create table customers(
custid int not null,
custname varchar(20),
address varchar(50) not null,
state varchar(10) not null,
primary key(custid));
```

```
insert into customers values(111,'elvin','202 jai street','delhi');
insert into customers values(113,'soman','puthumana','kerala');
insert into customers values(115,'mickey','juhu','maharastra');
insert into customers values(112,'patrick','harinagar','tamilnadu');
insert into customers values(114,'jaise','kottarakara','kerala');
```

0. Display the details customers table

```
select * from customers;
```

```
create table orders(
orderid int not null,
itemid int ,
quantity int not null,
orderdate date ,custid int
primary key(orderid),
foreign key(itemid) references items(itemid) on update cascade on delete
cascade ,
foreign key(custid) references customers(custid) on update cascade on delete
cascade));
```

```
insert into orders values(1,1,2,'2014-10-11',111);
insert into orders values(2,3,1,'2012-01-29',113);
insert into orders values(3,5,1,'2013-05-13',115);
```

```
insert into orders values(4,4,3,'2014-12-22',114);
```

0. Display the details orders table

```
select * from orders;
```

```
create table delivery(
deliveryid int not null,
orderid int ,custid int ,
primary key(deliveryid),
foreign key(orderid) references orders(orderid) on update cascade on delete
cascade,
foreign key(custid) references customers(custid) on update cascade on delete
cascade);
```

```
insert into delivery values(1001,1,111);
insert into delivery values(1002,2,113);
insert into delivery values(1003,3,115);
```

0. Display the details Delivery table

```
select * from delivery;
```

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

```
select customers.custid,custname,address,state from customers , orders
where orders.custid=customers.custid;
```

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

```
select customers.custid,custname,address,state from customers , delivery
where delivery.custid=customers.custid;
```

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

```
select orderdate from customers , orders
where orders.custid=customers.custid and custname like 'j%';
```

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

```
 select itemname,price from items as i ,customers as c,orders as o
where i.itemid=o.itemid and c.custid=o.custid and c.custname like'mickey';
```

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

```
 select c.* from customers as c ,orders as o
where o.custid=c.custid and orderdate>='2013-01-01' and c.custid not in
(select custid from delivery );
```

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

```
(select i.itemid from items as i ,orders as o where i.itemid=o.itemid)
union
(select i.itemid from items as i , orders as o where i.itemid=o.itemid
and o.orderid not in (select orderid from delivery) );
```

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

```
(select custname from customers as c,orders as o where o.custid=c.custid)
intersect
(select custname from customers as c,delivery as d where d.custid=c.custid);
```

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

```
(select custname from customers as c,orders as o where o.custid=c.custid)
except
(select custname from customers as c,delivery as d where d.custid=c.custid);
```

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

```
insert into orders values(5,2,1,'2012-05-25',115);

select * from customers where custid=(select custid from orders
group by custid order by count(*) desc LIMIT 1);
```

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

```
select c.* from customers as c,items as i,orders as o where o.itemid=i.itemid
and c.custid=o.custid and price>5000;
```

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

```
 (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='Samsung Galaxy S4' );
```

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

```
select * from customers left outer join orders on customers.custid=orders.custid;
select * from customers right outer join orders on customers.custid=orders.custid;
```

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

```
select count(*),state from customers group by state;
```

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

```
select * from items where price in (select price from items group by price
having price>(select avg(price) from items group by category));
```

8.4 Result

Implemented the programs using Join operation,set operation ,grouping and nested queries in Postgresql 11.5 (Ubuntu 11.5-1.pgdg18.04+1) in Ubuntu 18.04 and following output were obtained.

9 PL/PGSQL AND SEQUENCE

Exp No 8

9.1 Aim

To study the basic pl/pgsql and sequence queries.

9.2 Theory

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user-defined types, functions, and operators,
- can be defined to be trusted by the server, is easy to use.

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions. In PostgreSQL 9.0 and later, PL/pgSQL is installed by default. However it is still a loadable module, so especially security-conscious administrators could choose to remove it.

Sequences

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.

Without sequences, sequential values can only be produced programmatically. A new primary key value can be obtained by selecting the most recently produced value and incrementing it. This method requires a lock during the transaction and causes multiple users to wait for the next value of the primary key; this waiting is known as serialization. If you have such constructs in your applications, then you

should replace them with access to sequences. Sequences eliminate serialization and improve the concurrency of your application.

9.3 Questions

Write PL/SQL programs for the following:

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

```
CREATE or REPLACE FUNCTION prime(prime INT) RETURNS VOID as
$$
DECLARE
flag INT;
count INT =0;
i INT;
start INT=2;
rem INT;
BEGIN
WHILE (count<prime) LOOP
flag =0;
FOR i IN 2..(start/2) LOOP
rem=start%i;
IF  rem = 0 THEN
flag = 1;
END IF;
END LOOP;
IF flag = 0 THEN
RAISE NOTICE   ' % ' , start;
count=count+1;
END IF;
start=start +1;
END LOOP;END;
$$ LANGUAGE plpgsql;


select from prime(5);
```

9.3.2 Display the Fibonacci series upto 'n' terms

```
CREATE or REPLACE FUNCTION fib(fib INT) RETURNS VOID as
$$
DECLARE
value INT=1;
new INT=1;
temp INT;
count INT;
BEGIN
RAISE NOTICE ' %', value;
RAISE NOTICE ' %', new;
FOR count in 3..fib LOOP
temp=new;
new=new+value;
value=temp;
RAISE NOTICE ' %', new;
END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
   select * from fib(10);
```

9.3.3 Assigning Grade

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)

Table creation

```
 create table student_grade(roll int primary key,name varchar(10),mark1
int,mark2 int, mark3 int ,grade varchar(4));
insert into student_grade values(1,'anu',50,45,48),(2,'manu',50,50,50),
(3,'ramu',35,40,40);
```

Code

```
CREATE or REPLACE FUNCTION set_student_grade() RETURNS VOID as
$$
BEGIN
update student_grade
set grade='pass'
where (mark1+mark2+mark3)/3  >40;
update student_grade
set grade='fail'
where (mark1+mark2+mark3)/3 <=40;
END;
$$ LANGUAGE plpgsql;
```

Output

```
select from set_student_grade;
select * from student_grade;
```

9.3.4 Circle and Area

Create table circle_area (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.

Code

```
CREATE or REPLACE FUNCTION create_circle() RETURNS VOID as
$$
DECLARE
data1 INT =5;
data2 REAL;
count INT =5;
i INT ;
BEGIN
CREATE TABLE circle(rad INT ,area REAL);
FOR i IN 1..count LOOP
data2=3.1416*data1*data1;
INSERT INTO circle VALUES(data1,data2);
data1=data1+5;
END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Output

```
select from create_circle();
select * from circle;
```

9.3.5 Insertion using Array

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

```
create table stud(name varchar(10),mark int);
```

Code

```
CREATE or REPLACE FUNCTION insert_stud() RETURNS VOID as
$$
DECLARE
data1 INT []= '{ 25,76,43,45,67,57,97,56,89,8 }';
 data2  VARCHAR(20)[]='{ ARUN,AMAL,PETER,JOSE,ANNIE,MARY,JOSEPH,MARK,MIDHUN,KEVIN}'
i INT ;
BEGIN
FOR i IN 1..10 LOOP
INSERT INTO stud VALUES(data2[i],data1[i]);
END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Output

```
select from insert_stud();
select * from stud;
```

9.3.6 Insertion using Array

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

```
Create table class_cse (roll int primary key,name varchar(10),phone varchar(15));
```

Code

```
CREATE  SEQUENCE csekey
```

```
START 101;

CREATE or REPLACE FUNCTION class_cse() RETURNS VOID as
$$
DECLARE
data1 VARCHAR []= '{ 0482-239091,0484-234562 ,0485-11234,0489-43617,0481-23145}';
 data2  VARCHAR(20)[]='{ ARUN,AMAL,PETER,JOSE,ANNIE }';
i INT ;
j INT;
BEGIN
FOR i IN 1..5 LOOP
INSERT INTO class_cse VALUES(nextval('csekey'),data2[i],data1[i]);
END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Output

```
select from class_cse();
select * from class_cse;
```

9.4 Result

Implemented the PL/PGSQL programs in Postgresql 11.5 (Ubuntu 11.5-1.pgdg18.04+1) in Ubuntu 18.04 and following output were obtained.

10 CURSOR

Exp No 9

10.1 Aim

To study the use and implementation of cursors in PL/SQL.

10.2 Theory

A PL/PGSQL cursor allows us to encapsulate a query and process each individual row at a time. We use cursors when we want to divide a large result set into parts and process each part individually. If we process it at once, we may have a memory overflow error.

10.3 Questions

10.3.1 Grade calculation

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.

ID	SNAME	M1	M2	M3	GR
88	Anu	39	67	92	C
10	Jan	58	61	29	D
30	karuna	87	79	77	A
29	jossy	39	80	45	D

Table Creation

create table stdnt(id int,sname varchar(15),m1 int,m2 int,m3 int,gr char(1));

```
insert into stdnt (id,sname,m1,m2,m3) values(88,'anu',39,67,92);
insert into stdnt (id,sname,m1,m2,m3) values(10,'jan',58,61,29);
insert into stdnt (id,sname,m1,m2,m3) values(30,'karuna',87,79,77);
insert into stdnt (id,sname,m1,m2,m3) values(29,'jossy',39,80,45);
```

```
select * from stdnt;
```

10.3.2 Code

```
CREATE OR REPLACE FUNCTION get_grade()
   RETURNS void AS $$
DECLARE
    total INT ;
    grade char(1);
    rec_film   RECORD;
    cur_films CURSOR
       FOR SELECT * FROM stdnt;
BEGIN
   OPEN cur_films;
   LOOP
      FETCH cur_films INTO rec_film;
      EXIT WHEN NOT FOUND;
      total = rec_film.m1+rec_film.m2+rec_film.m3;
      IF total>240 THEN
          grade='A';
      ELSIF total>180 THEN
        grade='B';
      ELSIF total>120 THEN
        grade='C';
     ELSIF total>60 THEN
        grade='D';
    ELSE
            grade='F';
      END IF;
      update stdnt set gr=grade where m1=rec_film.m1;
   END LOOP;
   CLOSE cur_films;
END; $$
LANGUAGE plpgsql;
```

Output

```
select get_grade();
select * from stdnt;
```

10.3.3 Interest Calculation

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

ACCNO	NAME	BALANCE	ADATE
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

Table Creation

create table bankdetails(accno int,name varchar(15),balance int,adate date);
create table banknew(accno int,interest int);

```
insert into bankdetails values(1001,'aby',3005,'10-oct-15');
insert into bankdetails values(1002,'alan',4000,'05-may-95');
insert into bankdetails values(1003,'amal',5000,'16-mar-92');
insert into bankdetails values(1004,'jeffin',3500,'01-apr-50');
insert into bankdetails values(1005,'majo',6600,'01-jan-01');
```

```
select * from bankdetails;
```

Code

```
CREATE OR REPLACE FUNCTION get_interest()
   RETURNS void AS $$
DECLARE
    interest INT ;
    account   RECORD;
    movacc CURSOR
       FOR SELECT * FROM bankdetails;
```

```
BEGIN
   OPEN movacc;
   LOOP
      FETCH movacc INTO account;
      EXIT WHEN NOT FOUND;

      interest=0.08*account.balance;
      INSERT INTO banknew VALUES (account.accno,interest);
   END LOOP;
   CLOSE movacc;

END; $$

LANGUAGE plpgsql;
```

Output

```
select get_interest();
select * from banknew;
```

10.3.4 Finding Experienced People

Create table people_list (id, name, dt_joining, place).If person's experience is above 10 years, put the tuple in table exp_list (id, name, experience).)

ID	NAME	DT_JOINING	PLACE
102	Mathew	07-JUN-2008	CHY
103	Luffy	15-APR-2005	FSN
104	Lucci	13-AUG-2009	KTM
105	Law	12-APR-2005	WTC
101	Vivi	21-SEP-2010	ABA

Table creation

create table people_list(id INT, name varchar(20),dt_joining DATE,place varchar(20));

create table exp_list(id INT, name varchar(20),exp INT);

```
insert into people_list values(101,'Robert','03-APR-2005','CHY');
insert into people_list values(102,'Mathew','07-JUN-2008','CHY');
insert into people_list values(103,'Luffy','15-APR-2003','FSN');
insert into people_list values(104,'Lucci','13-AUG-2009','KTM');
insert into people_list values(105,'Law','14-APR-2005','WTC');
insert into people_list values(101,'Vivi','21-SEP-2010','ABA');


select * from people_list;
```

Code

```
CREATE OR REPLACE FUNCTION set_exp()
   RETURNS void AS $$
DECLARE
    exp INT;
    proff   RECORD;
    today DATE;
    movproff CURSOR
       FOR SELECT * FROM people_list;
BEGIN
   OPEN movproff;
   SELECT current_date INTO today;
   LOOP
      FETCH movproff INTO proff;
      EXIT WHEN NOT FOUND;

      SELECT DATE_PART('year', today::date) - DATE_PART('year', proff.dt_joining::d
      IF exp>10 THEN
       INSERT INTO exp_list VALUES (proff.id,proff.name,exp);
      END IF;
   END LOOP;
   CLOSE movproff;

END; $$

LANGUAGE plpgsql;
```

Output

```
select from set_exp;
select * from exp_list;
```

10.3.5 Salary Increment

Create table employee_list(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%.

ID	NAME	M_SAL
101	Mathew	55000
102	Jose	80000
103	John	250000
104	Ann	600000

Table creation

```
 create table emp_list(id INT,Name varchar(20),M_sal INT);

insert into emp_list values(101,'Mathew',55000);
insert into emp_list values(102,'Jose',80000);
insert into emp_list values(103,'John',250000);
insert into emp_list values(104,'Ann',600000);


   select * from emp_list;
```

Code

```
CREATE OR REPLACE FUNCTION sal_incre()
   RETURNS void AS $$
DECLARE
    yearsal INT;
    monsal INT;
```

```
    sal   RECORD;
    movsal CURSOR
       FOR SELECT * FROM emp_list;
BEGIN
   OPEN movsal;
   LOOP
      FETCH movsal INTO sal;
      EXIT WHEN NOT FOUND;
      yearsal=sal.m_sal*12;
      monsal=sal.m_sal;

        IF yearsal>500000 THEN
           UPDATE emp_list SET m_sal=monsal*1.1 WHERE m_sal=monsal;
        ELSIF yearsal>200000 THEN
        UPDATE emp_list SET m_sal=monsal*1.15 WHERE m_sal=monsal;
        ELSIF yearsal>60000 THEN
         UPDATE emp_list SET m_sal=monsal*1.2WHERE m_sal=monsal;
        ELSE
         UPDATE emp_list SET m_sal=monsal*1.25 WHERE m_sal=monsal;
     END IF;
   END LOOP;
   CLOSE movsal;

END; $$

LANGUAGE plpgsql;
```

Output

```
select from sal_incre();
select * from emp_list;
```

10.4 Result

Implemented the PL/PGSQL CURSOR programs in Postgresql 11.5 (Ubuntu 11.5-1.pgdg18.04+1) in Ubuntu 18.04 and following output were obtained.

11 TRIGGER AND EXCEPTION HANDLING

Exp No 10

11.1 Aim

To study PL/SQL trigger and exception handling.

11.2 Theory

TRIGGER

Triggers are procedures that are stored in the database and implicitly run, or fired, when something happens. Syntax for writing triggers is:

```
Create trigger <trigger name> Before/after/instead of

insert/delete/update

of <column name> on <table name>

for each row when <condition>

<pl/sql block with declare|begin--end constructs>
```

EXCEPTION

It is used to handle run time errors in program

Syntax:

```
begin

<executable statements> exception

<exception handling> end;
```

11.3 Questions

11.3.1 Trigger whenever data is inserted

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

Table Creation

```
 CREATE TABLE customer_details (cust_id int UNIQUE,cust_name varchar(25),address
varchar(30));
```

Code

```
CREATE OR REPLACE FUNCTION cust_det_insert() RETURNS TRIGGER AS
$cust_det_insert$
    BEGIN
     RAISE NOTICE 'A row is inserted';
RETURN NEW;
    END;
$cust_det_insert$
LANGUAGE plpgsql;
CREATE TRIGGER cust_det_insert
AFTER INSERT ON customer_details
    FOR EACH STATEMENT EXECUTE PROCEDURE cust_det_insert();
```

Output

```
INSERT INTO customer_details VALUES(1,'John','Ezhaparambbil');
```

11.3.2 Message when salary >20000

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

Table Creation

```
CREATE TABLE emp_details(empid INT UNIQUE,empname varchar(20),salary int);
```

Code

```
CREATE OR REPLACE FUNCTION emp_sal_check() RETURNS trigger AS $emp_sal$
    BEGIN
        IF NEW.salary >20000 THEN
           RAISE NOTICE 'Employee % has salary greater than 20000 ',NEW.empname;
        END IF;
        RETURN NEW;
    END;
$emp_sal$ LANGUAGE plpgsql;

CREATE TRIGGER emp_sal AFTER INSERT OR UPDATE ON emp_details
    FOR EACH ROW EXECUTE PROCEDURE emp_sal_check();
```

Output

```
INSERT INTO emp_details VALUES(1,'John',25000);
```

11.3.3 Row count

Create a trigger w.r.tcustomer_detailstable.
Increment the value of count_row (in cust_count table) whenever a new tuple is inserted and decrement the value of count_row when a tuple is deleted.
Initial value of the count_row is set to 0.

Table creation

```
CREATE TABLE cust_count(count_row int);
```

```
insert into cust_count VALUES(0);
```

Code

```
CREATE OR REPLACE FUNCTION cust_count() RETURNS trigger AS $cust_count$
DECLARE
count INT;
BEGIN
SELECT * FROM cust_count INTO count;
  IF (TG_OP = 'DELETE') THEN
IF count !=0 THEN
UPDATE  cust_count SET count_row=count_row-1;
END IF;
ELSIF (TG_OP = 'INSERT') THEN
UPDATE  cust_count SET count_row=count_row+1;
END IF;
RETURN NEW;
END;
$cust_count$ LANGUAGE plpgsql;

CREATE TRIGGER cust_count_change
AFTER INSERT OR DELETE ON customer_details
    FOR EACH ROW EXECUTE PROCEDURE cust_count();
```

11.3.4 Deletion and Updating

Create a trigger to insert the deleted rows from emp_details to another table and updated rows to another table. (Create the tables deleted and updatedT)

Table creation

```
CREATE TABLE deleted(empid INT ,empname varchar(20),salary int);
CREATE TABLE updated(empid INT,empname varchar(20),salary int);
```

Code

```
CREATE OR REPLACE FUNCTION del_upd() RETURNS trigger AS $del_upd$
BEGIN
```

```
  IF (TG_OP = 'DELETE') THEN
INSERT INTO deleted VALUES(OLD.empid,OLD.empname,OLD.salary);
ELSIF (TG_OP = 'UPDATE') THEN
     INSERT INTO updated VALUES(OLD.empid,OLD.empname,OLD.salary);
END IF;
RETURN NEW;
END;
$del_upd$ LANGUAGE plpgsql;

CREATE TRIGGER del_upd
AFTER UPDATE OR DELETE ON emp_details
    FOR EACH ROW EXECUTE PROCEDURE del_upd();
```

11.3.5 Divide zero Exception

Write a PL/SQL to show divide by zero exception

Code

```
CREATE OR REPLACE FUNCTION div(a INT,b INT) RETURNS INT as
$$
DECLARE
result INT;
BEGIN
IF b=0 THEN
RAISE EXCEPTION  'DVIDE BY ZERO  ' ;
ELSE
result=a/b;
RETURN result ;
END IF;
END;
$$
LANGUAGE plpgsql;
```

11.3.6 No Data Found Exception

Write a PL/SQL to show no data found exception

Code

```
CREATE OR REPLACE FUNCTION get_the_sal(id INT) RETURNS INT as
$$
```

```
DECLARE
result INT;
BEGIN
SELECT salary INTO result FROM emp_details WHERE empid=id;
IF RESULT IS NULL THEN
RAISE EXCEPTION 'NO DATA FOUND' ;
ELSE
RETURN result;
END IF;
END;
$$
LANGUAGE plpgsql;
```

11.3.7 Wrong Ebill

Create a table with ebill(cname,prevreading,currreading). If prevreading = currreading then raise an exception 'Data Entry Error'.

Table creation

CREATE TABLE ebill(cname varchar(20),preread int,curread int);

Code

```
CREATE OR REPLACE FUNCTION check_reading() RETURNS TRIGGER AS
$checkread$
BEGIN
IF NEW.preread=NEW.curread THEN
RAISE EXCEPTION 'DATA ENTRY ERROR A % B %' ,NEW.preread,NEW.curread;
ELSE
RAISE NOTICE 'STATEMENT PROCESSED'  ;
END IF;
RETURN NEW;
END;
$checkread$
LANGUAGE plpgsql;

CREATE TRIGGER check_reading
BEFORE INSERT ON ebill
```

```
FOR EACH ROW EXECUTE PROCEDURE check_reading();
```

11.4 Result

Implemented the PL/PGSQL programs for trigger and exception in Postgresql 11.5 (Ubuntu 11.5-1.pgdg18.04+1) in Ubuntu 18.04 and following output were obtained.

12 PROCEDURES , FUNCTIONS & SCHEMA

Exp No 11

12.1 Aim

To study PL/PGSQL Procedures , functions,schema.

12.2 Theory

FUNCTIONS

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

```
FUNCTION name [(parameter[, parameter, ...])]
    RETURN datatype IS [local declarations]
BEGIN
    executable statements
[EXCEPTION
    exception handlers]
END
[name];
```

PROCEDURES

A procedure is a subprogram that performs a specific action. You write procedures using the syntax

```
PROCEDURE name [(parameter[,parameter, ...])] IS [local declarations]
BEGIN
executable statements
[EXCEPTION
    exception handlers]
END
[name];
```

12.3 Questions

12.3.1 Factorial of a number

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

Code

```
CREATE OR REPLACE FUNCTION fact(fact INT) RETURNS INT AS
$$
DECLARE
    count INT = 1;
    result INT = 1;
BEGIN
    FOR count IN 1..fact LOOP
    result = result* count;
    END LOOP;
RETURN result;
END;
$$ LANGUAGE plpgsql;
```

12.3.2 Boost Marks

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

ROLL	MARKS	PHONE
1	70	9496947423
2	85	9495941358
3	78	8281865009

Table Creation

```
CREATE TABLE student_details(roll int,marks int, phone int);

INSERT INTO student_details VALUES(1,70,9496947423);
INSERT INTO student_details VALUES(2,85,9495941358);
INSERT INTO student_details VALUES(3,78,8281865009);
```

Code

```
CREATE OR REPLACE PROCEDURE boost()
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE student_details SET marks=marks*1.05;
END;
$$;
```

12.3.3 Finding Total and grade

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.

Table creation

CREATE TABLE studentmark(id int, name varchar(10), m1 int, m2 int, m3 int, total int, grade varchar(1));

Code

```
CREATE FUNCTION insert_stud(id INT,name varchar(20),m1 INT,m2 INT,m3 INT)
RETURNS VOID AS
$$
DECLARE
total INT;
grade CHAR;
BEGIN
total=m1+m2+m3;
INSERT INTO studentmark VALUES(id,name,m1,m2,m3,total);
IF  total >=240 THEN
grade='A';
ELSIF total >=180 THEN
grade='B';
ELSIF total>=120 THEN
grade='C';
```

```
ELSIF  total>=60 THEN
grade = 'D' ;
ELSE
grade ='F';
END IF;
CALL insert_grade(id,grade);
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE PROCEDURE insert_grade(sid INT ,sgrade CHAR)
LANGUAGE plpgsql
AS $$
BEGIN
UPDATE studentmark SET grade=sgrade WHERE id=sid;
END;
$$;
```

12.3.4 Schema

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

Code

```
CREATE SCHEMA pk1;

CREATE  OR REPLACE PROCEDURE pk1.proc1(num1 REAL,num2 REAL)
LANGUAGE plpgsql
AS
$$
DECLARE
sum REAL;
average REAL;
prod REAL;
```

```
BEGIN
sum = num1+num2;
prod = num1*num2;
average = (num1 + num2)/2;
RAISE NOTICE 'Sum of % and % is %' ,num1,num2,sum;
RAISE NOTICE 'Product of % and % is %' ,num1,num2 ,prod;
RAISE NOTICE 'Average of % and % is %' ,num1,num2,average;
END;
$$;

CREATE  OR REPLACE PROCEDURE pk1.proc2(num1 REAL)
LANGUAGE plpgsql
AS
$$
DECLARE
root REAL;
BEGIN
root=sqrt(num1);
RAISE NOTICE 'Root of % is %' ,num1,root;
END;
$$;
CREATE OR REPLACE FUNCTION pk1.fn11(num REAL) RETURNS VOID AS
$$
DECLARE
odd INT ;
BEGIN
odd = num;
odd=odd %2;
IF odd=1 THEN
RAISE NOTICE 'Number % is odd',num;
ELSE
RAISE NOTICE 'Number % is even',num;
END IF;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION pk1.fn22(num1 REAL,num2 REAL, num3 REAL) RETURNS VOID AS
$$
DECLARE
sum REAL ;
```

```
BEGIN
sum = num1+num2+num3;
RAISE NOTICE 'Sum of % ,%,% is %',num1,num2,num3,sum;
END;
$$ LANGUAGE plpgsql;


CREATE OR REPLACE FUNCTION pk1.all(num1 REAL,num2 REAL, num3 REAL)
 RETURNS VOID AS
$$
DECLARE
BEGIN
CALL pk1.proc1(num1,num2);
CALL pk1.proc2(num1);
PERFORM pk1.fn11(num1);
PERFORM pk1.fn22(num1,num2,num3);
END;
$$ LANGUAGE plpgsql;
```

12.4 Result

Implemented the PL/PGSQL programs using procedures , functions and schema in Postgresql 11.5 (Ubuntu 11.5-1.pgdg18.04+1) in Ubuntu 18.04 and following output were obtained.