EXPERIMENT 4	Basic SQL Commands
theory	SELECT: SELECT statement returns a result set of records from one or more tables. The select statement has optional clauses: WHERE specifies which rows to retrieve GROUP BY groups rows sharing a property so that an aggregate function can be applied to each group having group. HAVING selects among the groups defined by the GROUP BY clause. ORDER BY specifies an order in which to return the rows. Syntax: SELECT<attribute list> FROM<table list> WHERE<condition> Where Attribute list is a list of attribute name whose values to be retrieved by the query. Table list is a list of table name required to process query. Condition is a Boolean expression that identifies the tuples to be retrieved by query. SQL Aggregate Functions SQL aggregate functions return a single value, calculated from values in a column. Useful aggregate functions: AVG() - Returns the average value COUNT() - Returns the number of rows FIRST() - Returns the first value LAST() - Returns the last value MAX() - Returns the largest value MIN() - Returns the smallest value SUM() - Returns the sum The SQL ORDER BY Keyword The ORDER BY keyword is used to sort the result-set by one or more columns. The ORDER BY keyword sorts the records in ascending order by default. To sort the records in a descending order, you can

<table>
<tr><td></td><td>use the DESC keyword.
SQL ORDER BY Syntax
SELECT column_name, column_name
FROM table_name
ORDER BY column_name ASC|DESC, column_name
ASC|DESC;</td></tr>
<tr><td>Procedure</td><td>TASK 1:
1. Create following table:
Table name : sales_order
<table>
<tr><th>Column Name</th><th>Data type</th><th>Size</th></tr>
<tr><td>order_no</td><td>varchar</td><td>6</td></tr>
<tr><td>Order_date</td><td>date</td><td></td></tr>
<tr><td>Client_no</td><td>varchar</td><td>6</td></tr>
<tr><td>Dely_addr</td><td>varchar</td><td>25</td></tr>
<tr><td>Salesman_no</td><td>varchar</td><td>6</td></tr>
<tr><td>Dely_type</td><td>char</td><td>1</td></tr>
<tr><td>Billed_yn</td><td>char</td><td>1</td></tr>
<tr><td>Dely_date</td><td>Date</td><td></td></tr>
<tr><td>Order_status</td><td>varchar</td><td>10</td></tr>
</table>
2. Insert 5-6 records in table.
3. Find the names of all clients having ‘a’ as the second letter in their names.
4. Find out the clients who stay in a city whose second letter is ‘a’
5. Find the list of all clients who stay in ‘mumbai’ ordered by their names
6. Print the list of clients whose bal_due is greater than</td></tr>
</table>

	value 10000 7. Print the information from sales_order table for orders placed in the month of January 8. Display the order information for client_no C001 and C002 9. Find the products whose selling price is greater than 2000 and less than or equal to 5000 10. Find the products whose selling price is more than 1500. Calculate new selling price as original selling price * 1.5. Rename the new column in the above query as new_price 11. Count the total number of orders 12. Calculate the average price of all the product 13. Determine minimum and maximum product prices 14. count the number of products having price greater than or equal to 1500 15. Display the order number and day on which clients placed their order 16. Display the order_date in the format 'dd-month-yy' 17. Display the month (in alphabets) and date when the order must be delivered 18. Find the date, 15 days after today's date 19. Find the no. of days elapsed between today's date and the delivery date of orders placed by the clients. Task2: Use select with where statement with SQL aggregate functions for the tables created in Expt. no. 3

	EXP 5A
Theory	**SQL:** It is structured query language, basically used to pass the query to retrieve and manipulate the information from database **DDL:** The Data Definition Language (DDL) is used to create the database (i.e. tables, keys, relationships etc), maintain the structure of the database and destroy databases and database objects. Eg. Create, Drop, Alter, Describe, Truncate 1. **CREATE statements: It is used to create the table.** CREATE TABLE table_name(columnName1 datatype(size), columnName2 datatype(size),.....); 2. **DROP statements:** To destroy an existing database, table, index, or view. If a table is dropped all records held within it are lost and cannot be recovered. DROP TABLE table_name; 3. **ALTER statements:** To modify an existing database object. **Adding new columns:** Alter table table_name Add(New_columnName1 datatype(size), New_columnName2 datatype(size),.....); **Dropping a columns from a table** : Alter table table_name DROP column columnName: **Modifying Existing columns:** Alter table table_name Modify (columnName1 Newdatatype(Newsize)); 4. **Describe statements:** To describe the structure (column and data types) of an existing database, table, index, or view. DESC table_name; 5. **Truncate statements:** To destroy the data in an existing database, table, index, or view. If a table is truncated all records held within it are lost and cannot be recovered but the table structure is maintained. TRUNCATE TABLE table_name;
Procedure	1. Write a query to create a table employee with empno, ename, designation, and salary. Emp (empno number (4), ename

	varchar2 (10), designatin varchar2 (10), salary number (8,2)); 2. Write a Query to Alter the column empno number (4) to empno number (6). 3. Write a Query to Alter the table employee with multiple columns (empno, ename.) 4. Write a query to add a new column in to employee as qualification varchar2(6) 5. Write a query to add multiple columns in to employee dob date , doj date 6. Write a query to drop a column ‘doj’ from an existing table employee 7. Write a query to drop multiple columns ‘dob’ and ‘qualification’ from employee 8. Truncate table EMP 9. Drop table EMP

Experiment No. 5 B- Integrity Constraints

AIM:

- To implement database for relational model using DDL statement
- Apply Integrity Constraints for the specified system

Objective of the Experiment:

After completing this experiment you will be able to:

Create database.

Create table with constraints

Modify the schema of the table.

Theory :

Pre Lab/ Prior Concepts:

The Data Definition Language (DDL) is used to create and modify the relational schema. Also it is used to add various constraints to the table like the primary key, foreign key, check constraint, not null constraint and unique constraint.

The DDL statements are:

CREATE

DROP

ALTER

SQL supports the standard int, smallint, real, double precision, char(N), varchar(N), date, time, timestamp, and interval for creating tables.

Procedure / Algorithm:

Create Database and use it:

$ createdb mydb

$ psql mydb

Delete a database:

$ dropdb mydb

Create table:

```
CREATE TABLE my_first_table
( first_column text,
second_column integer
);

CREATE TABLE products (
product_no integer,
name text, price numeric);
```

Drop Table:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Default Value:

```
CREATE TABLE products (
product_no integer,
name text,
price numeric DEFAULT 9.99 );
```

Constraints:

1. Primary Key

```
CREATE TABLE products (
```

product_no integer **PRIMARY KEY**,

name text,

price numeric);

Primary keys can also constrain more than one column.

CREATE TABLE example (

a integer,

b integer,

c integer,

PRIMARY KEY (a, c)

);

2. Check Constraint

CREATE TABLE products (

product_no integer,

name text,

price numeric **CHECK (price > 0)**);

3. Not Null Constraint

CREATE TABLE products (

product_no integer **NOT NULL**,

name text **NOT NULL**,

price numeric);

4. Unique Constraint

CREATE TABLE products (

product_no integer **UNIQUE**,

name text,

price numeric);

5. Foreign Key Constarint

```
CREATE TABLE products (
product_no integer PRIMARY KEY,
name text,
price numeric
```

CREATE TABLE orders (

order_id integer PRIMARY KEY,

product_no integer **REFERENCES products (product_no)**,

quantity integer

);

Here a foreign key constraint in the order table references the products table.

Modifying table:

Adding column

ALTER TABLE products ADD COLUMN description text;

Removing column

ALTER TABLE products DROP COLUMN description;

Adding Constraint

ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no); ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;

Removing Constraint

ALTER TABLE products DROP CONSTRAINT some_name;

Adding Not Null Constraint

ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;

Removing Not Null Constraint

ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;

Task2: Use the concepts of Primary KEY and foreign key for converting ER diagram of ur Project in relational model.

Conclusion:

Thus using the schema diagram from the previous experiment, the tables were created using CREATE DDL statement with the primary key and foreign key constraint. Other constraints like Check, Unique and Not Null were added to the appropriate column by using ALTER DDL statement

EXPERIMENT 6	Complex SQL commands

Theory

Joining Tables

The FROM clause allows more than 1 table in its list, however simply listing more than one table will *very* rarely produce the expected results. The rows from one table must be correlated with the rows of the others. This correlation is known as *joining*.

In the subsequent text, the following 3 example tables are used:

p Table (parts)

pno	descr	color
P1	Widget	Blue
P2	Widget	Red
P3	Dongle	Green

s Table (suppliers)

sno	name	city
S1	Pierre	Paris
S2	John	London
S3	Mario	Rome

sp Table (suppliers & parts)

sno	pno	qty
S1	P1	NULL
S2	P1	200
S3	P1	1000
S3	P2	200

An example can best illustrate the rationale behind joins. The following query:

SELECT * FROM sp, p

Produces:

sno	pno	qty	pno	descr	color
S1	P1	NULL	P1	Widget	Blue
S1	P1	NULL	P2	Widget	Red
S1	P1	NULL	P3	Dongle	Green
S	P	20	P	Widg	Blu

2	1	0	1	et	e
S2	P1	200	P2	Widget	Red
S2	P1	200	P3	Dongle	Green
S3	P1	1000	P1	Widget	Blue
S3	P1	1000	P2	Widget	Red
S3	P1	1000	P3	Dongle	Green
S3	P2	200	P1	Widget	Blue
S3	P2	200	P2	Widget	Red
S3	P2	200	P3	Dongle	Green

Each row in *sp* is arbitrarily combined with each row in *p*, giving 12 result rows (4 rows in *sp* X 3 rows in *p*.) This is known as a *cartesian product*.

A more usable query would correlate the rows from *sp* with rows from *p*, for instance matching on the common column -- *pno*:

SELECT *
FROM sp, p
WHERE sp.pno = p.pno

This produces:

sno	**pno**	**qty**	**pno**	**descr**	**color**
S1	P1	NULL	P1	Widget	Blue
S2	P1	200	P1	Widget	Blue
S3	P1	1000	P1	Widget	Blue
S3	P2	200	P2	Widget	Red

More information refer this
https://www.tutorialspoint.com/sql/sql-using-joins.ht

	m
Procedure	1. Create following table: Table name : sales_order_details

Column Name	Data type	Size
order_no	varchar	6
Product_no	varchar	6
Qty_ordered	numeric	8
Qty_disp	numeric	8
Product_rate	numeric	10,2

Create table- customer(cid, cname, address, pno)

Create table- cust_order(cid foreign key, order_no foreign key)

2. Insert 5-6 records in table in each tables.
3. Print the description and total qty sold for each product
4. Find the value of each product sold
5. Calculate the average quantity sold for each client that has a maximum order value of 15000
6. find out the sum total of all the billed orders for the month of January
7. find out the name of customers who have given the order of more than 10 qty.
8. find out the customer names with product no with maximum qty ordered.

EXPERIMENT 7	Nested subqueries in SQL
Aim	To implement nested sub-queries in SQL
Procedure	Perform the following queries using nested sub-queries 1. Find the product no. and description of non-moving products i.e. products not being sold. 2. Find the customer name, address for the client who has placed order no 'O191' 3. Find the clients names who have placed orders before the month of May'96 4. Find out if the product '1.44 Drive' has been ordered by any client and print the client_no, name to whom it was sold 5. Find the names of clients who have placed orders worth Rs. 10000 or more 6. Retrieve all the orders placed by a client named 'Rahul Desai' from the sales_order table. 7. Find out all the products that are not being sold from the product_master table, based on the products actually sold as shown in the sales_order_details table. 8. Retrieve the product numbers, their description and the total quantity ordered for each product.

EXPERIMENT:-8

Procedure

PL/pgSQL is a loadable procedural language for the Postgres database system. This package was originally written by Jan Wieck. 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.

Structure of PL/pgSQL

PL/pgSQL is a block-structured language. The complete text of a function definition must be a block. A block is defined as:

```
[<<label>>]
[ DECLARE Declarations ]
BEGIN
statements
END [label];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END , as shown above; however the final END that concludes a function body does not require a semicolon

```
IF boolean-expression THEN
statements
END IF;
```

```
IF boolean-expression
THEN
statements
ELSE statements
END IF;
```

```
WHILE
boolean-expression
LOOP
statements
END LOOP [label];
```

```
FOR name IN [ REVERSE ]
expression..expression
[ BY expression] LOOP
statements
END LOOP [label];
FOR i IN 1..10 LOOP
-- i will take on the values 1,2,3,4,5,6,7,8,9,10
within the loop
END LOOP;
FOR i IN REVERSE 10..1 LOOP
-- i will take on the values 10,9,8,7,6,5,4,3,2,1
within the loop
END LOOP;
FOR i IN REVERSE 10..1 BY 2 LOOP
-- i will take on the values 10,8,6,4,2 within
the loop
END LOOP;
```

EXPERIMENT 09	***Functions and Triggers***
Aim	To implement PL/pgSQL function and trigger
Theory	CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION will either create a new function, or replace an existing definition. To be able to define a function, the user must have the USAGE privilege on the language. If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema. The name of the new function must not match any existing function with the same input argument types in the same schema. However, functions of different argument types can share a name (this is called *overloading*). **Syntax for Function** CREATE [OR REPLACE] FUNCTION name ([[argmode] [argname] argtype [{ DEFAULT | = } default_expr] [, ...]]) [RETURNS rettype | RETURNS TABLE (column_name column_type [, ...])] { LANGUAGE lang_name | WINDOW | IMMUTABLE | STABLE | VOLATILE | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER | COST execution_cost | ROWS result_rows | SET configuration_parameter { TO value | = value | FROM CURRENT } | AS 'definition' | AS 'obj_file', 'link_symbol' } ... [WITH (attribute [, ...])] If you drop and then recreate a function, the new function is not the same entity as the old; you will have to drop existing rules, views, triggers, etc. that refer to the old function. Use CREATE OR REPLACE FUNCTION to change a function definition without breaking objects that refer to the function. The trigger can be specified to fire before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE, or DELETE is attempted); or after the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed); or instead of the operation (in the case of inserts, updates or deletes on a view). If the trigger fires before or instead of the event, the trigger can skip the operation for the current row, or change the row being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the effects of other triggers, are "visible" to the trigger.
	Syntax of Trigger CREATE [CONSTRAINT] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [OR ...] } ON table

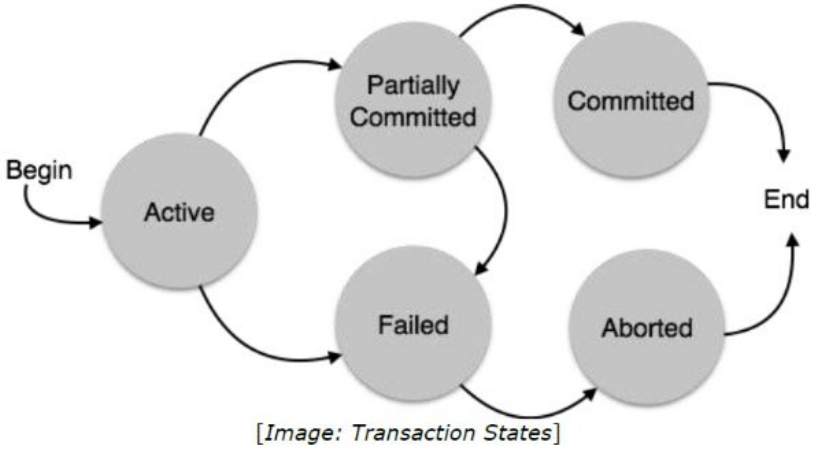
	[FROM referenced_table_name] [NOT DEFERRABLE | [DEFERRABLE] { INITIALLY IMMEDIATE | INITIALLY DEFERRED }] [FOR [EACH] { ROW | STATEMENT }] [WHEN (condition)] EXECUTE PROCEDURE function_name (arguments) where event can be one of: INSERT UPDATE [OF column_name [, ...]] DELETE TRUNCATE To create a trigger on a table, the user must have the TRIGGER privilege on the table. The user must also have EXECUTE privilege on the trigger function. Use DROP TRIGGER to remove a trigger.
Procedure	1. Write a function to find factorial of a number 2. Create table emp(id,name,salary) and insert 3 records in it. 3. Write a function find average salary from emp table 4. Write a row level trigger that would fire before insert/ update/delete operations performed on emp table, not allowing these operations and display the appropriate message. 5. Write a row level trigger that would fire after insert/update/delete operations performed on emp table displaying date on which data manipulation performed.

EXPERIMENT 10	***Transaction concept***
Aim	To implement Simple Transaction concept
Tools	PostgreSQL

Theory	A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.
	Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.
	Properties of Transactions
	Transactions have the following four standard properties, usually referred to by the acronym ACID −
	• **Atomicity** − Ensures that all operations within the work unit are completed successfully; otherwise, the transaction is aborted at the point of failure and previous operations are rolled back to their former state.
	• **Consistency** − Ensures that the database properly changes states upon a successfully committed transaction.
	• **Isolation** − Enables transactions to operate independently of and transparent to each other.
	• **Durability** − Ensures that the result or effect of a committed transaction persists in case of a system failure.
	
	In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with BEGIN and COMMIT commands. So our banking transaction would actually look like:
	BEGIN;
	UPDATE accounts SET balance = balance - 100.00
	WHERE name = 'Alice';
	-- etc etc
	COMMIT;
	End;

Theory	**State Diagram :**

A transaction in a database can be in one of the following states:



[*Image: Transaction States*]

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account.

```
BEGIN;
--sql
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Bob';
ROLLBACK TO my_savepoint; or commit;
--UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Wally';
COMMIT;
```

Theory	# Transaction Control The following commands are used to control transactions − • **BEGIN TRANSACTION** − To start a transaction. • **COMMIT** − To save the changes, alternatively you can use **END TRANSACTION** command. • **ROLLBACK** − To rollback the changes. Transactional control commands are only used with the DML commands INSERT, UPDATE and DELETE only. They cannot be used while creating tables or dropping them because these operations are automatically committed in the database. ## The BEGIN TRANSACTION Command Transactions can be started using BEGIN TRANSACTION or simply BEGIN command. Such transactions usually persist until the next COMMIT or ROLLBACK command is encountered. But a transaction will also ROLLBACK if the database is closed or if an error occurs. **The following is the simple syntax to start a transaction −** **BEGIN;** **or** **BEGIN TRANSACTION;** **The COMMIT Command** **The COMMIT command is the transactional command used to save changes invoked by a transaction to the database.** **The COMMIT command saves all transactions to the database since the last COMMIT or ROLLBACK command.** **The syntax for COMMIT command is as follows −** **COMMIT;** **or** **END TRANSACTION;**

Theory	**The ROLLBACK Command** The ROLLBACK command is the transactional command used to undo transactions that have not already been saved to the database. The ROLLBACK command can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued. The syntax for ROLLBACK command is as follows − ROLLBACK;
Task	**WA java program using JDBC to implement transaction** **http://www.postgresqltutorial.com/postgresql-jdbc/transaction/**
Links	**https://www.postgresql.org/docs/9.1/sql-start-transaction.html** **https://www.postgresql.org/docs/8.3/tutorial-transactions.html** **https://pgdash.io/blog/postgres-transactions.html** **https://tapoueh.org/blog/2018/07/postgresql-concurrency-isolation-and-locking/**