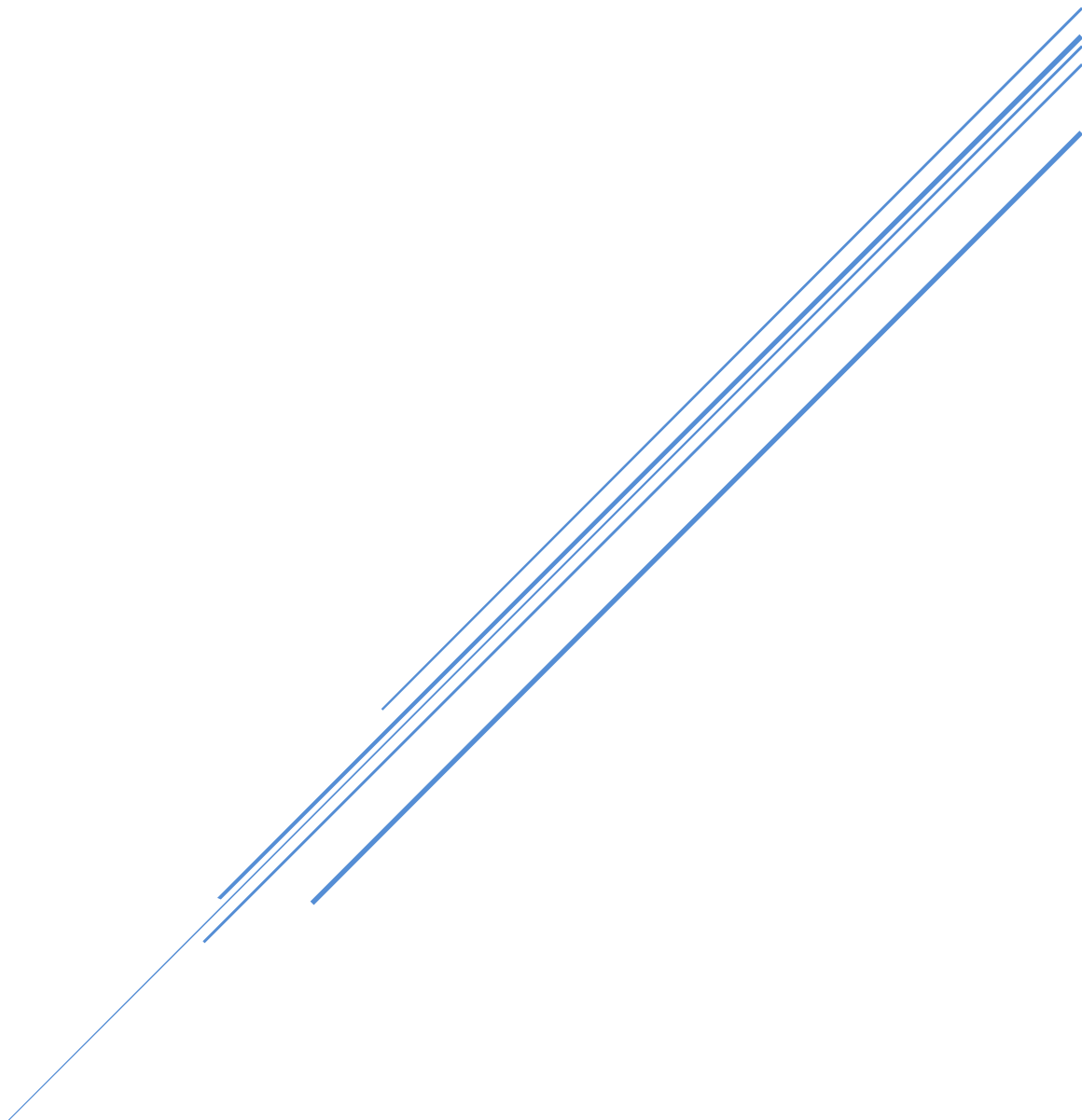


POSTGRESQL SQL SCRIPTING AND ANALYTICS



John Rey Goh
2024

Contents

Introduction to PostgreSQL and Intermediate SQL	3
Review of basic SQL concepts	13
Introduction to PostgreSQL architecture.....	5
Setting up the PostgreSQL environment	4
Connecting to the database using psql and pgAdmin	11
Advanced SELECT Statements.....	18
Subqueries and correlated subqueries	18
Common Table Expressions (CTEs)	19
Window functions (ROW_NUMBER, RANK, DENSE_RANK, etc.).....	20
Joins and Set Operations.....	21
Advanced JOIN techniques (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN)	21
Using UNION, INTERSECT, and EXCEPT	22
Advanced Data Types and Functions	24
Arrays, JSON, and JSONB.....	24
Date and time functions	26
String functions and text search	26
Indexes and Performance Tuning	27
Types of indexes (B-tree, Hash, GIN, GiST)	27
Creating and managing indexes.....	29
Analyzing query performance with EXPLAIN	30
Advanced Query Optimization.....	31
Understanding the PostgreSQL query planner	31
Analyzing query plans	31
Techniques for query optimization.....	32
Stored Procedures and Functions	33
Creating and using stored procedures.....	33
Writing functions in PL/pgSQL	35
Advanced use of PL/pgSQL features	37
Triggers and Event-Driven Programming.....	40
Creating and managing triggers.....	40
Writing trigger functions.....	41
Using triggers for data validation and automation.....	44
Transactions and Concurrency Control.....	49
Understanding transactions and ACID properties	49
Using SAVEPOINT and ROLLBACK	51
Managing concurrency with locking mechanisms	52

Security and User Management 53

 Role-based access control..... 53

 Managing permissions and roles 54

 Best practices for database security 55

Introduction to Analytics with PostgreSQL..... 56

 Overview of analytical capabilities in PostgreSQL 56

 Setting up a data warehouse environment 58

 ETL processes using PostgreSQL 59

Advanced Analytical Queries 61

 Using window functions for analytics 61

 Aggregate functions and GROUP BY extensions..... 62

 Pivoting and unpivoting data 64

Data Visualization and Reporting..... 66

 Integrating PostgreSQL with BI tools 66

 Creating views for reporting 67

 Using PostgreSQL with Python for data analysis and visualization 68

Time-Series and Geospatial Data 70

 Working with time-series data in PostgreSQL 70

 Introduction to PostGIS for geospatial data 72

 Advanced queries with time-series and geospatial data 73

Introduction to PostgreSQL and Intermediate SQL

Training Lab Setup

- ✓ Download and install PostgreSQL 15/16 <https://www.postgresql.org/download/>

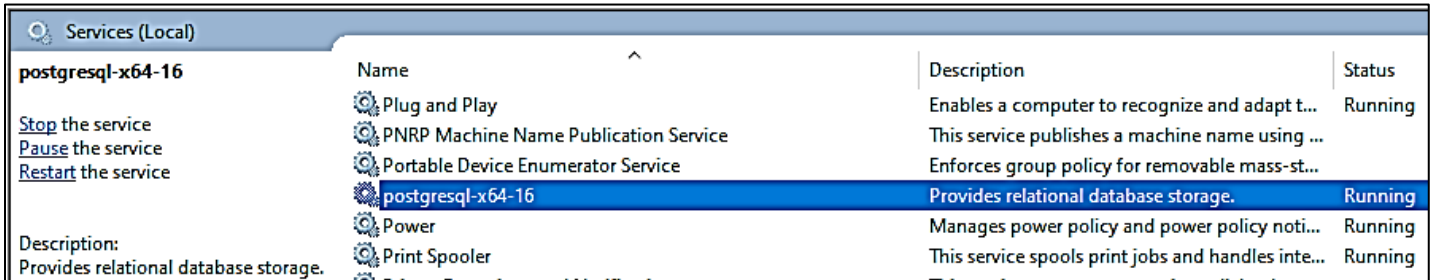
PostgreSQL History Notes

- ✓ PostgreSQL is a free and open-source object -relational database management system (ORDBMS)
- ✓ Developers can use OOP techniques involving Classes, Methods and Objects with PostgreSQL
- ✓ Began in 1986 as Postgres in University of California in Berkeley
- ✓ Michael Stonebraker and friends Developed Postgres
- ✓ PostgreSQL is cross-platform
- ✓ PostgreSQL features transactions with Atomicity, Consistency, Isolation, Durability (ACID) properties
- ✓ PostgreSQL manages concurrency through multiversion concurrency control (MVCC)

Installation Notes

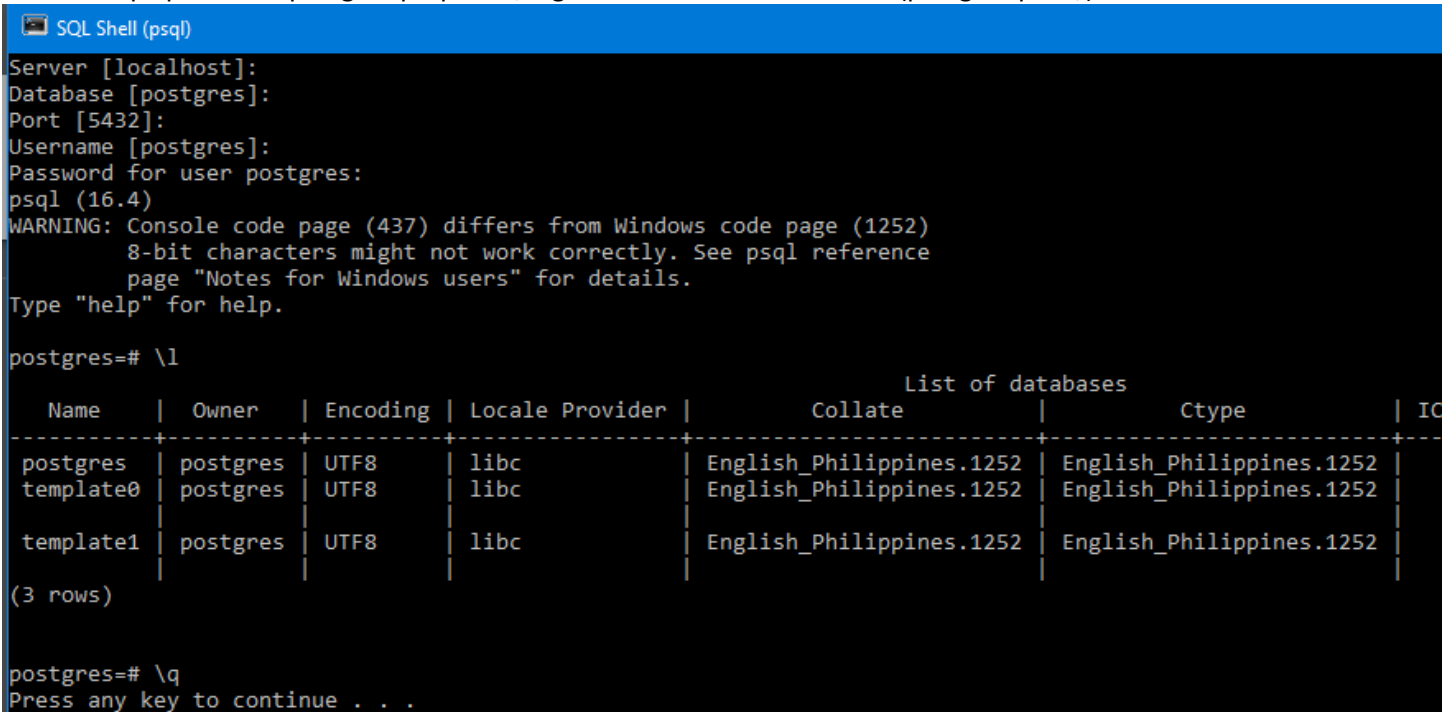
- Follow with the steps in the installation wizard while discussing options (Install pgAdmin also and StackBuilder).
Note of the Password and Port used.
- PostgreSQL runs as a service
- To verify:

Windows Services → postgresql [version]



Services (Local)			
	Name	Description	Status
Stop the service Pause the service Restart the service Description: Provides relational database storage.	Plug and Play	Enables a computer to recognize and adapt t...	Running
	PNRP Machine Name Publication Service	This service publishes a machine name using ...	
	Portable Device Enumerator Service	Enforces group policy for removable mass-st...	
	postgresql-x64-16	Provides relational database storage.	Running
	Power	Manages power policy and power policy noti...	Running
	Print Spooler	This service spools print jobs and handles inte...	Running

Search for psql to enter postgresql sql shell, login then list down databases (postgresql=# \l)



```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres:
psql (16.4)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

postgres=# \l

              List of databases
  Name      | Owner   | Encoding | Locale Provider | Collate          | Ctype            | IC
-----+-----+-----+-----+-----+-----+-----
 postgres   | postgres | UTF8     | libc             | English_Philippines.1252 | English_Philippines.1252 |
 template0  | postgres | UTF8     | libc             | English_Philippines.1252 | English_Philippines.1252 |
 template1  | postgres | UTF8     | libc             | English_Philippines.1252 | English_Philippines.1252 |
(3 rows)

postgres=# \q
Press any key to continue . . .
```

Setting up the PostgreSQL environment

Setup PostgreSQL Environment Variable on Windows

System properties → environment variables → add to path:
C:\Program Files\PostgreSQL\12\bin

Add a new system variable

Variable Name: PGDATA
Variable Value: C:\Program Files\PostgreSQL\12\data

View help and other options

```
\> psql --help
```

Check version

```
\> psql -V  
\> psql --version
```

Connect and open sql shell

```
\> psql -U postgres -h localhost -p 5432 -d postgres
```

List databases

```
\> psql -U postgres -h localhost -l
```

a. note in linux (ex. RedHat)

1. Download postgresql and install using rpm command
2. Check if 'postgres' user has been created and set password to it

```
# su postgres  
$ exit  
# passwd postgres
```

3. Manage profile

```
# su postgres  
# cd ~  
# vim .bash_profile
```

...

```
PATH=$PATH:HOME/bin  
export PATH  
export PATH=/usr/pgsql-16/bin:$PATH  
PG_DATA=/var/lib/pgsql/16/data  
export PGDATA
```

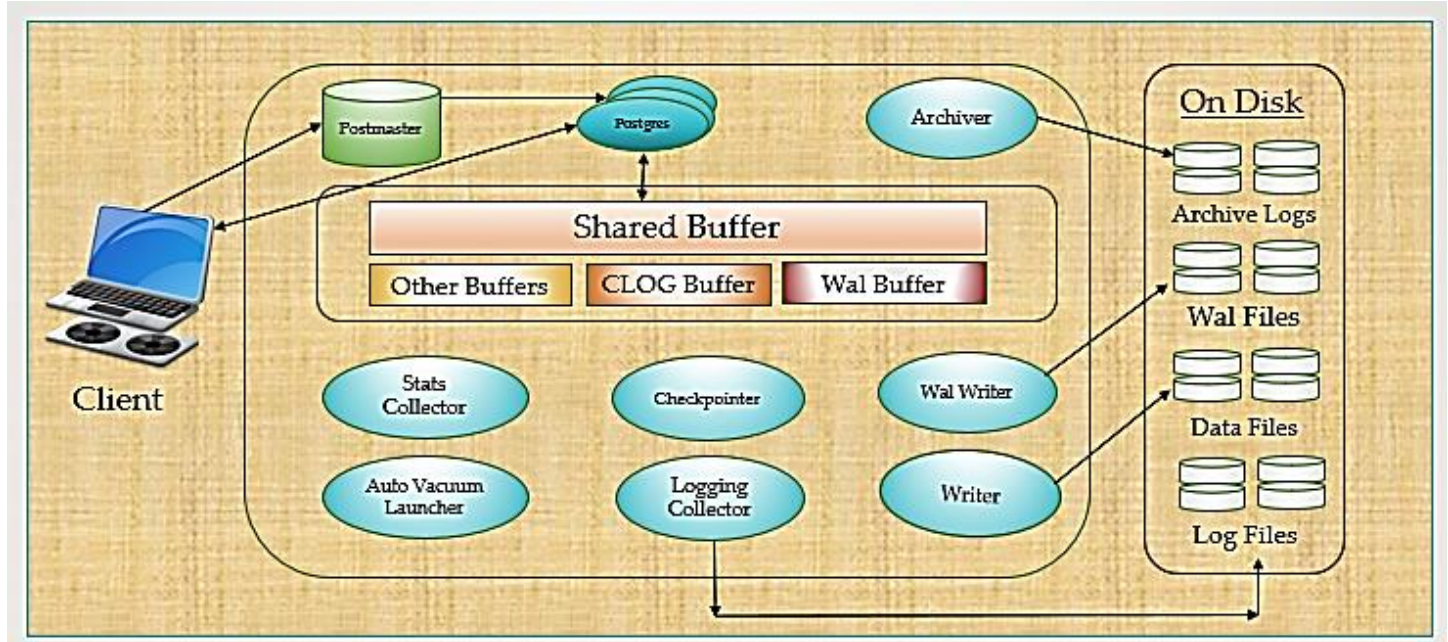
4. Check and manage services

```
# systemctl status postgresql-16  
# systemctl start postgresql-16  
# systemctl stop postgresql-16
```

Introduction to PostgreSQL architecture

Process and Memory Architecture

- PostgreSQL is a relational database management system with a client-server architecture. Many clients receive and send request to centralized database server.
- PostgreSQL uses "process per-user" client/server model. Each user is granted a process.
- PostgreSQL has a set of processes and memory structures which constitutes an instance.
- Programs run by clients connect to the server instance and request read and write operations.
- Default port of PostgreSQL is 5432.



Postmaster Process

- Postmaster is the first process which gets started in PostgreSQL.
- Postmaster acts as supervisor process, whose job is to monitor, start, restart some processes if they die.
- Postmaster acts as a listener and receive new connection request from the client.
- Postmaster is responsible for Authentication and Authorization of all incoming request.
- Postmaster spawns a new process called Postgres for each new connection.

Utility Processes

- Bgwriter\Writer** : Periodically writes the dirty buffer to a data file.
- Wal Writer** : Write the WAL buffer to the WAL file.
- Checkpoint** : Checkpoint is invoked every 5 minute(default) or when max_wal_size value is exceeded. The checkpoint syncs all the buffers from the shared buffer area to the data files.
- Auto vacuum** : Responsible to carry vacuum operations on bloated tables.(If Enabled).
- Statscollector** : Responsible for collection and reporting of information about server activity then update the information to optimizer dictionary((pg_catalog)).
- Logwriter\Logger** : Write the error message to the log file.
- Archiver (Optional)** :When in Archive.log mode, copy the WAL file to the specified directory.

Memory Segments

- Shared Buffers
- Wal Buffers
- Clog Buffers
- Work Memory
- Maintenance Work Memory
- Temp Buffers

Shared Buffer:

- ✓ User cannot access the datafile directly to read or write any data.
- ✓ Any select, insert, update or delete to the data is done via shared buffer area.
- ✓ The data that is written or modified in this location is called "Dirty data".
- ✓ Dirty data is written to the data files located in physical disk through background writer process.
- ✓ Shared Buffers are controlled by parameter named: shared buffer located in postgresql.conf file.

Wal Buffer:

- ✓ Write Ahead Logs buffer is also called as "Transaction Log Buffers".
- ✓ WAL data is the metadata information about changes to the actual data, and is sufficient to reconstruct actual data during database recovery operations.
- ✓ WAL data is written to a set of physical files in persistent location called "WAL segments" or "checkpoint segments".
- ✓ Wal buffers are flushed from the buffer area to wal segments by wal writer.
- ✓ Wal buffers memory allocation is controlled by the wal_buffers parameter.

Clog and other buffers:

- ✓ CLOG stands for "commit log", and the CLOG buffers is an area in operating system RAM dedicated to hold commit log pages.
- ✓ The commit logs have commit status of all transactions and indicate whether or not a transaction has been completed (committed).
- ✓ Work Memory is a memory reserved for either a single sort or hash table (Parameter: Work_mem)
- ✓ Maintenance Work Memory is allocated for Maintenance work (Parameter: maintenance_work_mem).
- ✓ Temp Buffers are used for access to temporary tables in a user session during large sort and hash table. (Parameter: temp_buffers).

Physical Files in PostgreSQL

Physical Files:

- Data Files: It is a file which is used to store data. It does not contain any instructions or code to be executed.
- Wal Files: Write ahead log file, where all transactions are written first before commit happens.
- Log Files: All server messages, including stderr, csvlog and syslog are logged in log files.
- Archive Logs(Optional): Data from wal segments are written on to archive log files to be used for recovery purpose.

PostgreSQL Cluster (Initdb, Start/Stop/Restart/Reload)

Database Cluster:

- Database cluster is a collection of databases that is managed by a single instance on a server.
- Initdb creates a new PostgreSQL database cluster.
- Creating a database cluster consists of creating the directories in which the data is store. We call this the "data directory".
- We have to first initialize the storage area on the disk before we begin any operation on the database.

- Location of Data Directory:

Linux: /var/lib/pgsql/data (Not mandatory)

Windows: C:\Program Files\PostgreSQL\12\data (Not mandatory)

Initdb:

- We have to be logged in as PostgreSQL user (Linux) to execute the below commands.
- There are two way to initialize database.
- Syntax:
- `initdb -D /usr/local/pgsql/data(Linux)`
- `initdb -D -U postgres /usr/local/pgsql/data(Windows)`
- `pg_ctl -D -U postgres /usr/local/pgsql/data initdb`
- -D = refers to the data directory location.
- -W = we can use this option to force the super user to provide password before initialize db

Start\Stop Cluster:

- Start Cluster Syntax:

Linux

```
# systemctl start postgresql-12
```

Windows

```
\> pg_ctl -D "C:\Program Files\PostgreSQL\12\data" start
```

- Stop Cluster Syntax:

Linux

```
# systemctl stop postgresql-12
```

Windows

```
\> pg_ctl stop -D "C:\Program Files\Postgresql \12\data" -m shutdown mode
```

Types of Shutdown:

- Smart: the server disallows new connections, but let's existing sessions end their work normally. It shuts down only after all of the sessions terminate
- Fast :(Default): The server disallows new connections and abort their current transactions and exits gracefully.
- Immediate: Quits/aborts without proper shutdown which lead to recovery on next startup.

Difference between Reload and Restart.

- When we make changes to server parameters, we need to reload the configuration for them to take effect.
- Reload will just reload the new configurations, without restarting the service.
- Few configuration changes in server parameters, Do not get reflected until we restart the service.
- Restart gracefully shutdown all activity, relinquishes the resource, close all open files and start again with new configuration.

Reload\Restart Cluster:

- Syntax for Restart of Cluster:

Linux

```
# system reload postgresql-11
```

Windows

```
\> pg_ctl reload
```

- Syntax for Reload of Cluster:

Linux

```
# systemctl restart postgresql-11
```

Windows

```
\> pg_ctl restart
```

- Psql Command line:

```
SQL: SELECT pg_reload_conf(); (Irrespective of Env)
```

Pg_Controldata:

- Pg_controldata – Information about cluster.

Linux Syntax

```
./pg_controldata /var/lib/pgsqli/12/data/
```

```
[postgres@pgsql01 ~]$ ./pg_controldata /var/lib/pgsqli/11/data/
pg_control version number:          1100
Catalog version number:            201809051
Database system identifier:         6827166080220811381
Database cluster state:             in production
pg_control last modified:           Wed 03 Jun 2020 04:16:59 PM EDT
Latest checkpoint location:         0/EE8EBC0
Latest checkpoint's REDO location:  0/EE8EBC0
Latest checkpoint's REDO WAL file:  000000010000000000000000E
Latest checkpoint's TimeLineID:     1
Latest checkpoint's PrevTimeLineID: 1
Latest checkpoint's full_page_writes: on
```

Creating a Database

Create database Psql / createdb utility:

Syntax from psql:

```
create database databasename owner ownername;
```

Syntax from command line:

```
\> createdb <dbname>.
```

Syntax for help:

```
createdb --help
```

Drop database – Psql/ dropdb utility:

We can't drop the database which we are connected.

Example:

```
scott=# drop database scott;  
ERROR: cannot drop the currently open database
```

Syntax from psql:

```
Drop database <dbname>.
```

Syntax from command line:

```
\>dropdb <dbname>.
```

Syntax for dropdb help:

```
dropdb -help
```

Creating a user and granting / revoking privileges

Create user – Psql/ createuser utility/ Interactive:

Syntax from psql:

```
create user scott login superuser password 'welcome';
```

Syntax from command line:

```
\> createuser <username>
```

Syntax for interactive user creation from command line: Example:

```
createuser --interactive joe  
Shall the new role be a superuser? (y/n) n  
Shall the new role be allowed to create databases? (y/n) y  
Shall the new role be allowed to create more new roles? (y/n) y
```

Syntax for createuser help:

```
createuser --help
```

Drop user - Psql/ dropuser utility:

Syntax from psql:

```
drop user <username>
```

Syntax from command line:

```
\> dropuser <username>
```

Dropping a user with objects or privileges will return an error.

Example:

```
postgres=# drop user test1;  
ERROR: role "test1" cannot be dropped because some objects depend on it
```

Assign the user privileges to another user before dropping the user.

Example:

```
REASSIGN OWNED BY user to postgres;  
Drop role <username>;
```

Grant:

Grant CONNECT to the database:

```
GRANT CONNECT ON DATABASE database_name TO username;
```

Grant USAGE on schema:

```
GRANT USAGE ON SCHEMA schema_name TO username;
```

Grant on all tables for DML statements: SELECT, INSERT, UPDATE, DELETE

```
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA schema_name TO username;
```

Grant all privileges on all tables in the schema:

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA schema_name TO username;
```

Grant all privileges on all sequences in the schema:

```
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA schema_name TO username;
```

Grant permission to create database:

```
ALTER USER username CREATEDB;
```

Make a user superuser:

```
ALTER USER myuser WITH SUPERUSER;
```

Remove superuser status:

```
ALTER USER username WITH NOSUPERUSER;
```

Column Level access:

```
GRANT SELECT (col1), UPDATE (col1) ON mytable TO user;
```

Revoke Examples

Revoke Delete/update privilege on table from user

```
REVOKE DELETE, UPDATE ON products FROM user;
```

Revoke all privilege on table from user

```
REVOKE ALL ON products FROM user;
```

Revoke select privilege on table from all users (Public)

```
REVOKE SELECT ON products FROM PUBLIC;
```

Connecting to the database using psql and pgAdmin

Using psql

Connect to Specific Database with user and password

```
\> psql -d database -U user -W (-d =Database, -U = User, -W = Password)
```

Connect to Database on a different host/machine.

```
\> psql -h host -d database -U user -W
```

Connect using SSL Mode

```
\> psql -U user -h host "dbname=db sslmode=require"
```

Switch connection to a new database

```
postgres=# \c test1  
You are now connected to database "test1" as user "postgres".
```

List available databases

```
postgres=# \l
```

List available tables

```
postgres=# \dt
```

List available tables in all schemas

```
postgres=# \dt *.
```

Describe a table

```
postgres=# \d table_name
```

List available schema (+ to get more info)

```
postgres=# \dn
```

List available functions(+ to get more info)

```
postgres=# \df
```

List available views(+ to get more info)

```
postgres=# \dv
```

List users and their roles(+ to get more info)

```
postgres=# \du
```

List available sequence(+ to get more info)

```
postgres=# \ds
```

Execute the previous command

```
postgres=# \g
```

Command history

```
postgres=# \s
```

Save Command History to file:

```
postgres=# \s filename
```

Get help on psql commands

```
postgres=# \?
```

Turn on\off query execution time

```
postgres=# \timing
```

Edit statements in editor

```
postgres=# \e
```

Edit Functions in editor

```
postgres=# \ef
```

set output from non-aligned to aligned column output.

```
postgres=# \a
```

Formats output to HTML format.

```
postgres=# \H
```

Connection Information

```
postgres=# \conninfo
```

Quit psql

```
postgres=# \q
```

Run sql statements from a file.

```
\> psql -d test1 -U test1 -f test1.sql
```

Send the output to a file.

```
postgres=# \o test.txt
```

←set the output file first

```
postgres=# \l
```

←the result of this query will no longer appear in command window, look for the file

```
postgres=# \q
```

Save query buffer to filename.

```
postgres=# \w filename
```

Turn off auto commit on session level

```
\set AUTOCOMMIT off
```

Creating and Dropping Schema

Create & Drop Schema

Create Schema

```
CREATE schema <schema_name>;
```

Create Schema for a user, the schema will also be named as the user

```
Create schema authorization <username>;
```

Create Schema named John, that will be owned by brett

```
CREATE schema IF NOT EXISTS john AUTHORIZATION brett;
```

Drop a Schema

```
Drop schema <schema_name>;
```

(We cannot drop schema if there are any object associate with it.)

Schema Search Path:

Show search path can be used to find the current search path. Example:

```
postgres=# show search_path;
```

```
search_path
-----
"$user", public
( 1 row)
```

Default "\$user" is a special option that says if there is a schema that matches the current user (i.e SELECT SESSION_USER;), then search within that schema.

Search path can be set at session level, user level, database level and cluster level. Example:

```
Test1=# SET search_path TO test1,public;
```

```
Test1=# \dt
```

List of relations

Schema	Name	Type	Owner
--------	------	------	-------

test1	abc	table	test1
-------	-----	-------	-------

(1 rows)

Review of basic SQL concepts

Structured Query Language (SQL) is the standard language for interacting with databases. Basic concepts include:

- CREATE: Used to create new objects like table and schema
- ALTER: Modify properties of existing objects like tables and schema
- SELECT: Used to query data.
- INSERT: Adds new rows to a table.
- UPDATE: Modifies existing data.
- DELETE: Removes rows from a table.
- JOIN: Combines data from two or more tables based on a related column.

Examples:

Create Tables

We will create two tables: employeework and employeepersonal.

```
-- Create employeework table
CREATE TABLE employeework (
  id SERIAL PRIMARY KEY,
  position VARCHAR(50),
  department VARCHAR(50),
  datehired DATE,
  monthsalary NUMERIC(10, 2)
);

-- Create employeepersonal table
CREATE TABLE employeepersonal (
  id SERIAL PRIMARY KEY,
  firstname VARCHAR(50),
  lastname VARCHAR(50),
  address TEXT,
  birthdate DATE,
  civilstatus VARCHAR(20)
);
```

Insert Records into Each Table

```
-- Insert records into employeework table
INSERT INTO employeework (position, department, datehired, monthsalary)
VALUES
('Software Engineer', 'IT', '2020-01-15', 5000.00),
('Data Analyst', 'IT', '2021-03-10', 4500.00),
('HR Manager', 'HR', '2019-06-20', 6000.00),
('Accountant', 'Finance', '2018-12-11', 5200.00),
('Marketing Manager', 'Marketing', '2021-04-01', 4800.00),
('Sales Executive', 'Sales', '2017-09-12', 4700.00),
('Project Manager', 'IT', '2019-02-15', 7000.00),
('Recruiter', 'HR', '2020-08-05', 4000.00),
('Business Analyst', 'Finance', '2020-05-25', 5300.00),
('UX Designer', 'IT', '2021-09-23', 5500.00);

-- Insert records into employeepersonal table
INSERT INTO employeepersonal (firstname, lastname, address, birthdate, civilstatus)
VALUES
('John', 'Doe', '123 Elm Street', '1990-05-10', 'Single'),
('Jane', 'Smith', '456 Oak Street', '1985-07-22', 'Married'),
('Mark', 'Johnson', '789 Pine Avenue', '1992-12-15', 'Single'),
('Emily', 'Clark', '101 Maple Drive', '1995-03-30', 'Married'),
('Michael', 'Wilson', '202 Cedar Lane', '1988-11-05', 'Single'),
('Sarah', 'Lee', '303 Birch Boulevard', '1991-06-18', 'Married'),
('David', 'Anderson', '404 Spruce Road', '1994-09-27', 'Single'),
('Jessica', 'Taylor', '505 Cherry Street', '1989-02-14', 'Married'),
('Daniel', 'Thomas', '606 Poplar Circle', '1993-10-20', 'Single'),
('Samantha', 'Harris', '707 Willow Lane', '1990-01-25', 'Divorced');
```

Select Statements

Select All Records:

```
SELECT * FROM employeework;
```

Select Specific Columns:

```
SELECT firstname, lastname FROM employeepersonal;
```

Select with WHERE Condition:

```
SELECT * FROM employeepersonal WHERE civilstatus = 'Married';
```

Select Using Aliases:

```
SELECT firstname AS "First Name", lastname AS "Last Name" FROM employeepersonal;
```

Select with ORDER BY:

```
SELECT * FROM employeework ORDER BY monthsalary DESC;
```

Select with LIMIT:

```
SELECT * FROM employeepersonal LIMIT 5;
```

Select DISTINCT:

```
SELECT DISTINCT department FROM employeework;
```

Select with BETWEEN:

```
SELECT * FROM employeework WHERE monthsalary BETWEEN 4500 AND 6000;
```

Select with IN Clause:

```
SELECT * FROM employeepersonal WHERE civilstatus IN ('Married', 'Single');
```

Select with LIKE:

```
SELECT * FROM employeepersonal WHERE address LIKE '%Street%';
```

Select with INNER JOIN:

```
SELECT ep.firstname, ep.lastname, ew.position, ew.department  
FROM employeepersonal ep  
INNER JOIN employeework ew ON ep.id = ew.id;
```

Select with LEFT JOIN:

```
SELECT ep.firstname, ep.lastname, ew.position  
FROM employeepersonal ep  
LEFT JOIN employeework ew ON ep.id = ew.id;
```

Select with RIGHT JOIN:

```
SELECT ep.firstname, ew.position  
FROM employeepersonal ep  
RIGHT JOIN employeework ew ON ep.id = ew.id;
```


Select with Aggregate Function (COUNT):

```
SELECT department, COUNT(*) AS total_employees
FROM employeework
GROUP BY department;
```

Select with Aggregate Function (AVG):

```
SELECT department, AVG(monthlysalary) AS avg_salary
FROM employeework
GROUP BY department;
```

Select with Aggregate Function (SUM):

```
SELECT department, SUM(monthlysalary) AS total_salary
FROM employeework
GROUP BY department;
```

Select with HAVING Clause:

```
SELECT department, AVG(monthlysalary) AS avg_salary
FROM employeework
GROUP BY department
HAVING AVG(monthlysalary) > 5000;
```

Select with Subquery:

```
SELECT firstname, lastname FROM employeepersonal
WHERE id IN (SELECT id FROM employeework WHERE department = 'IT');
```

Select with CASE Statement:

```
SELECT firstname, lastname,
CASE
  WHEN civilstatus = 'Married' THEN 'Yes'
  ELSE 'No'
END AS "Is Married"
FROM employeepersonal;
```

Select with COALESCE:

```
SELECT firstname, lastname, COALESCE(civilstatus, 'Unknown') AS civil_status
FROM employeepersonal;
```

UPDATE and DELETE SQL Statements

Update a Single Record:

```
UPDATE employeework SET monthlysalary = 5500 WHERE id = 2;
```

Update Multiple Records:

```
UPDATE employeepersonal SET civilstatus = 'Single' WHERE birthdate > '1990-01-01';
```

Update with JOIN:

```
UPDATE employeework ew
SET department = 'Operations'
FROM employeepersonal ep
WHERE ew.id = ep.id AND ep.lastname = 'Wilson';
```

Delete a Single Record:

```
DELETE FROM employeepersonal WHERE id = 5;
```

Delete Records with a Condition:

```
DELETE FROM employeework WHERE monthsalary < 5000;
```

JOINS

Inner Join:

```
SELECT ep.firstname, ep.lastname, ew.position, ew.department
FROM employeepersonal ep
INNER JOIN employeework ew ON ep.id = ew.id;
```

Left Join:

```
SELECT ep.firstname, ep.lastname, ew.position
FROM employeepersonal ep
LEFT JOIN employeework ew ON ep.id = ew.id;
```

Right Join:

```
SELECT ep.firstname, ew.position
FROM employeepersonal ep
RIGHT JOIN employeework ew ON ep.id = ew.id;
```

Cross Join:

```
SELECT ep.firstname, ew.position
FROM employeepersonal ep
CROSS JOIN employeework ew;
```

Full Outer Join:

```
SELECT ep.firstname, ep.lastname, ew.position, ew.department
FROM employeepersonal ep
FULL OUTER JOIN employeework ew ON ep.id = ew.id;
```

Advanced SELECT Statements

Subqueries and correlated subqueries

Simple Subquery in SELECT:

```
SELECT firstname, lastname  
FROM employeepersonal  
WHERE id IN (SELECT id FROM employeework WHERE department = 'IT');
```

Subquery in WHERE:

```
SELECT * FROM employeework  
WHERE monthsalary > (SELECT AVG(monthsalary) FROM employeework);
```

Subquery with EXISTS:

```
SELECT firstname, lastname  
FROM employeepersonal  
WHERE EXISTS (SELECT 1 FROM employeework WHERE employeework.id = employeepersonal.id);
```

Subquery with NOT EXISTS:

```
SELECT firstname, lastname  
FROM employeepersonal  
WHERE NOT EXISTS (SELECT 1 FROM employeework WHERE employeepersonal.id = employeework.id);
```

Subquery with IN:

```
SELECT firstname, lastname  
FROM employeepersonal  
WHERE id IN (SELECT id FROM employeework WHERE monthsalary > 5000);
```

Subquery with ALL:

```
SELECT firstname, lastname  
FROM employeepersonal  
WHERE id = ALL (SELECT id FROM employeework WHERE department = 'Finance');
```

Subquery with Aggregate Function:

```
SELECT department, (SELECT AVG(monthsalary) FROM employeework WHERE department = ew.department)  
FROM employeework ew  
GROUP BY department;
```

Correlated Subquery in WHERE:

```
SELECT firstname, lastname  
FROM employeepersonal ep  
WHERE birthdate = (SELECT MAX(birthdate) FROM employeepersonal WHERE civilstatus = ep.civilstatus);
```

Correlated Subquery with EXISTS:

```
SELECT firstname, lastname  
FROM employeepersonal ep  
WHERE EXISTS (SELECT 1 FROM employeework ew WHERE ew.id = ep.id AND ew.monthsalary > 5000);
```

Correlated Subquery with UPDATE:

```
UPDATE employeework ew
SET department = 'HR'
WHERE EXISTS (SELECT 1 FROM employeepersonal ep WHERE ew.id = ep.id AND ep.civilstatus = 'Single');
```

Common Table Expressions (CTEs)

Basic CTE:

```
WITH employee_cte AS (
  SELECT id, firstname, lastname FROM employeepersonal
)
SELECT * FROM employee_cte;
```

CTE with Aggregation:

```
WITH department_salaries AS (
  SELECT department, SUM(monthlysalary) AS total_salary FROM employeework GROUP BY department
)
SELECT * FROM department_salaries;
```

CTE with Join:

```
WITH employee_details AS (
  SELECT ep.firstname, ep.lastname, ew.department
  FROM employeepersonal ep
  JOIN employeework ew ON ep.id = ew.id
)
SELECT * FROM employee_details;
```

Recursive CTE:

```
WITH RECURSIVE employee_hierarchy AS (
  SELECT id, firstname, lastname, NULL AS manager_id FROM employeepersonal WHERE id = 1
  UNION ALL
  SELECT ep.id, ep.firstname, ep.lastname, eh.id
  FROM employeepersonal ep
  JOIN employee_hierarchy eh ON ep.id = eh.manager_id
)
SELECT * FROM employee_hierarchy;
```

CTE with Filtering:

```
WITH high_salary_cte AS (
  SELECT id, monthlysalary FROM employeework WHERE monthlysalary > 5000
)
SELECT * FROM high_salary_cte WHERE id < 5;
```

Multiple CTEs:

```
WITH salary_cte AS (  
    SELECT department, AVG(monthlysalary) AS avg_salary FROM employeework GROUP BY department  
)  
high_salary_cte AS (  
    SELECT department FROM salary_cte WHERE avg_salary > 5000  
)  
SELECT * FROM high_salary_cte;
```

CTE with Subquery:

```
WITH department_high_salaries AS (  
    SELECT department, SUM(monthlysalary) AS total_salary FROM employeework GROUP BY department  
)  
SELECT * FROM department_high_salaries WHERE total_salary > (SELECT AVG(total_salary) FROM  
department_high_salaries);
```

CTE with Update:

```
WITH high_salary_employees AS (  
    SELECT id FROM employeework WHERE monthlysalary > 6000  
)  
UPDATE employeework SET department = 'Executive' WHERE id IN (SELECT id FROM high_salary_employees);
```

CTE with Deletion:

```
WITH old_employees AS (  
    SELECT id FROM employeework WHERE datehired < '2010-01-01'  
)  
DELETE FROM employeework WHERE id IN (SELECT id FROM old_employees);
```

CTE with Window Function:

```
WITH employee_rankings AS (  
    SELECT id, monthlysalary, ROW_NUMBER() OVER (ORDER BY monthlysalary DESC) AS rank FROM employeework  
)  
SELECT * FROM employee_rankings WHERE rank <= 5;
```

Window functions (ROW_NUMBER, RANK, DENSE_RANK, etc.)

ROW_NUMBER:

```
SELECT id, monthlysalary, ROW_NUMBER() OVER (ORDER BY monthlysalary DESC) AS row_num  
FROM employeework;
```

RANK:

```
SELECT id, monthlysalary, RANK() OVER (ORDER BY monthlysalary DESC) AS rank  
FROM employeework;
```

DENSE_RANK:

```
SELECT id, monthlysalary, DENSE_RANK() OVER (ORDER BY monthlysalary DESC) AS dense_rank  
FROM employeework;
```

NTILE:

```
SELECT id, monthsalary, NTILE(4) OVER (ORDER BY monthsalary DESC) AS salary_quartile
FROM employeeework;
```

LEAD:

```
SELECT id, monthsalary, LEAD(monthsalary, 1) OVER (ORDER BY monthsalary DESC) AS next_salary
FROM employeeework;
```

LAG:

```
SELECT id, monthsalary, LAG(monthsalary, 1) OVER (ORDER BY monthsalary DESC) AS previous_salary
FROM employeeework;
```

First_Value:

```
SELECT id, monthsalary, FIRST_VALUE(monthsalary) OVER (ORDER BY monthsalary DESC) AS highest_salary
FROM employeeework;
```

Last_Value:

```
SELECT id, monthsalary, LAST_VALUE(monthsalary) OVER (ORDER BY monthsalary ASC) AS lowest_salary
FROM employeeework;
```

CUME_DIST:

```
SELECT id, monthsalary, CUME_DIST() OVER (ORDER BY monthsalary DESC) AS cumulative_distribution
FROM employeeework;
```

PERCENT_RANK:

```
SELECT id, monthsalary, PERCENT_RANK() OVER (ORDER BY monthsalary DESC) AS percent_rank
FROM employeeework;
```

Joins and Set Operations

Advanced JOIN techniques (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN)

Basic INNER JOIN:

```
SELECT ep.firstname, ep.lastname, ew.position
FROM employeeepersonal ep
INNER JOIN employeeework ew ON ep.id = ew.id;
```

LEFT JOIN:

```
SELECT ep.firstname, ep.lastname, ew.position
FROM employeeepersonal ep
LEFT JOIN employeeework ew ON ep.id = ew.id;
```

RIGHT JOIN:

```
SELECT ep.firstname, ew.position
FROM employeeepersonal ep
RIGHT JOIN employeeework ew ON ep.id = ew.id;
```

FULL OUTER JOIN:

```
SELECT ep.firstname, ew.position
FROM employeepersonal ep
FULL OUTER JOIN employeework ew ON ep.id = ew.id;
```

Self Join:

```
SELECT e1.firstname AS "Employee", e2.firstname AS "Manager"
FROM employeepersonal e1
JOIN employeepersonal e2 ON e1.id = e2.id;
```

Cross Join:

```
SELECT ep.firstname, ew.position
FROM employeepersonal ep
CROSS JOIN employeework ew;
```

Join with Subquery:

```
SELECT ep.firstname, ew.department
FROM employeepersonal ep
JOIN (SELECT id, department FROM employeework WHERE monthsalary > 5000) ew
ON ep.id = ew.id;
```

Join with Aggregation:

```
SELECT ew.department, COUNT(ep.id) AS employee_count
FROM employeepersonal ep
JOIN employeework ew ON ep.id = ew.id
GROUP BY ew.department;
```

Join with Window Function:

```
SELECT ep.firstname, ew.department, RANK() OVER (PARTITION BY ew.department ORDER BY ew.monthsalary DESC)
AS rank
FROM employeepersonal ep
JOIN employeework ew ON ep.id = ew.id;
```

Join Multiple Tables:

```
SELECT ep.firstname, ep.lastname, ew.position, ew.department
FROM employeepersonal ep
JOIN employeework ew ON ep.id = ew.id
JOIN another_table at ON at.id = ew.id;
```

Using UNION, INTERSECT, and EXCEPT

UNION Example:

```
SELECT firstname FROM employeepersonal
UNION
SELECT lastname FROM employeepersonal;
```

UNION ALL Example:

```
SELECT firstname FROM employeepersonal  
UNION ALL  
SELECT lastname FROM employeepersonal;
```

INTERSECT Example:

```
SELECT firstname FROM employeepersonal  
INTERSECT  
SELECT lastname FROM employeepersonal;
```

EXCEPT Example:

```
SELECT firstname FROM employeepersonal  
EXCEPT  
SELECT lastname FROM employeepersonal;
```

UNION with WHERE:

```
SELECT firstname FROM employeepersonal WHERE civilstatus = 'Married'  
UNION  
SELECT lastname FROM employeepersonal WHERE civilstatus = 'Married';
```

INTERSECT with WHERE:

```
SELECT firstname FROM employeepersonal WHERE birthdate > '1990-01-01'  
INTERSECT  
SELECT lastname FROM employeepersonal WHERE birthdate > '1990-01-01';
```

EXCEPT with WHERE:

```
SELECT firstname FROM employeepersonal WHERE civilstatus = 'Single'  
EXCEPT  
SELECT lastname FROM employeepersonal WHERE civilstatus = 'Single';
```

UNION with ORDER BY:

```
SELECT firstname FROM employeepersonal  
UNION  
SELECT lastname FROM employeepersonal  
ORDER BY firstname;
```

INTERSECT with Subquery:

```
SELECT firstname FROM employeepersonal  
INTERSECT  
SELECT firstname FROM employeepersonal WHERE id IN (SELECT id FROM employeework WHERE monthsalary >  
5000);
```

EXCEPT with Subquery:

```
SELECT firstname FROM employeepersonal  
EXCEPT  
SELECT firstname FROM employeepersonal WHERE id IN (SELECT id FROM employeework WHERE monthsalary >  
5000);
```


Advanced Data Types and Functions

Arrays, JSON, and JSONB

In PostgreSQL, arrays allow you to store multiple values in a single field. Arrays can store any data type, including integers, text, and more. Array functions and operators help manipulate array data.

Arrays Example

Create a table with an array field:

```
CREATE TABLE employee_skills (  
  id SERIAL PRIMARY KEY,  
  name VARCHAR(50),  
  skills TEXT[]  
);
```

Insert data into the array field:

```
INSERT INTO employee_skills (name, skills)  
VALUES ('John Doe', ARRAY['Java', 'SQL', 'Python']);
```

Select an element from an array:

```
SELECT skills[1] FROM employee_skills WHERE name = 'John Doe';
```

Check if a value exists in an array:

```
SELECT * FROM employee_skills WHERE 'SQL' = ANY (skills);
```

Select all rows where array contains a specific value:

```
SELECT * FROM employee_skills WHERE ARRAY['SQL'] <@ skills;
```

Update an array:

```
UPDATE employee_skills SET skills = ARRAY_APPEND(skills, 'C++') WHERE name = 'John Doe';
```

Concatenate two arrays:

```
SELECT ARRAY[1, 2, 3] || ARRAY[4, 5, 6] AS concatenated_array;
```

Remove an element from an array:

```
UPDATE employee_skills SET skills = ARRAY_REMOVE(skills, 'SQL') WHERE name = 'John Doe';
```

Length of an array:

```
SELECT array_length(skills, 1) FROM employee_skills WHERE name = 'John Doe';
```

Unnest an array (convert array elements into rows):

```
SELECT unnest(skills) FROM employee_skills WHERE name = 'John Doe';
```

JSON and JSONB are data types in PostgreSQL to store JSON data. JSONB stores data in a binary format, offering better performance for querying and indexing than JSON.

JSON/JSONB Examples:

Create a table with JSON/JSONB fields:

```
CREATE TABLE employee_info (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR(50),  
    details JSONB  
);
```

Insert JSONB data:

```
INSERT INTO employee_info (name, details)  
VALUES ('Alice', '{"age": 30, "department": "IT", "skills": ["Python", "SQL"]}');
```

Select specific JSON elements:

```
SELECT details->'age' AS age FROM employee_info WHERE name = 'Alice';
```

Select nested JSON elements:

```
SELECT details->'skills'->>1 AS second_skill FROM employee_info WHERE name = 'Alice';
```

Filter rows by JSON field:

```
SELECT * FROM employee_info WHERE details->>'department' = 'IT';
```

Update a JSON field:

```
UPDATE employee_info SET details = jsonb_set(details, '{age}', '31') WHERE name = 'Alice';
```

Check if a key exists in JSON:

```
SELECT * FROM employee_info WHERE details ? 'age';
```

Remove a key from JSONB:

```
UPDATE employee_info SET details = details - 'skills' WHERE name = 'Alice';
```

Concatenate JSON objects:

```
SELECT '{"name": "Alice"}'::jsonb || '{"age": 30}'::jsonb;
```

Convert JSONB to text:

```
SELECT details::text FROM employee_info WHERE name = 'Alice';
```

Date and time functions

PostgreSQL provides extensive support for date and time functions, including extraction, comparison, and manipulation of dates and times.

Date and Time Functions Examples:

Get current date:

```
SELECT CURRENT_DATE;
```

Get current time:

```
SELECT CURRENT_TIME;
```

Extract year from a date:

```
SELECT EXTRACT(YEAR FROM '2023-09-14'::DATE);
```

Add interval to a date:

```
SELECT '2023-09-14'::DATE + INTERVAL '1 year';
```

Subtract interval from a timestamp:

```
SELECT '2023-09-14 10:00:00'::TIMESTAMP - INTERVAL '2 days';
```

Calculate the difference between two dates:

```
SELECT AGE('2023-09-14'::DATE, '1990-01-01'::DATE);
```

Format date output:

```
SELECT TO_CHAR(NOW(), 'YYYY-MM-DD');
```

Get the day of the week:

```
SELECT EXTRACT(DOW FROM '2023-09-14'::DATE); -- 0 for Sunday, 6 for Saturday
```

Get the start of the month:

```
SELECT DATE_TRUNC('month', '2023-09-14'::DATE);
```

Check if a timestamp is between two times:

```
SELECT '2023-09-14 10:00:00'::TIMESTAMP BETWEEN '2023-09-01 00:00:00'::TIMESTAMP AND '2023-09-30 23:59:59'::TIMESTAMP;
```

String functions and text search

PostgreSQL supports a variety of string functions and text search capabilities for manipulating and querying string data.

String Functions & Text Search Examples:

Concatenate strings:

```
SELECT 'Hello' || ' ' || 'World';
```

Convert string to upper case:

```
SELECT UPPER('postgresql');
```

Convert string to lower case:

```
SELECT LOWER('PostgreSQL');
```

Extract substring:

```
SELECT SUBSTRING('PostgreSQL' FROM 5 FOR 4); -- Extract 'greS'
```

Replace part of a string:

```
SELECT REPLACE('Hello World', 'World', 'PostgreSQL');
```

Position of substring:

```
SELECT POSITION('SQL' IN 'PostgreSQL');
```

Length of a string:

```
SELECT LENGTH('PostgreSQL');
```

Trim spaces:

```
SELECT TRIM(' PostgreSQL ');
```

Search for a text pattern using LIKE:

```
SELECT * FROM employeepersonal WHERE lastname LIKE 'Sm%';
```

Full-text search using to_tsvector and to_tsquery:

```
SELECT * FROM employeepersonal  
WHERE to_tsvector(firstname || ' ' || lastname) @@ to_tsquery('John & Doe');
```

Indexes and Performance Tuning

Types of indexes (B-tree, Hash, GIN, GiST)

Indexes in PostgreSQL improve the speed of data retrieval operations by creating a smaller, more searchable structure based on specific columns. PostgreSQL supports several types of indexes, each suited for different data types and query patterns.

B-tree Index

The B-tree index is the default index type in PostgreSQL, used for equality, range queries, and sorting. It works well with most data types.

Create a B-tree index:

```
CREATE INDEX idx_name_btree ON employeepersonal (lastname);
```

Query that uses a B-tree index:

```
SELECT * FROM employeepersonal WHERE lastname = 'Doe';
```

B-tree index with multiple columns:

```
CREATE INDEX idx_employee_btree ON employeepersonal (lastname, firstname);
```

Unique B-tree index:

```
CREATE UNIQUE INDEX idx_unique_btree ON employeepersonal (id);
```

Drop a B-tree index:

```
DROP INDEX idx_name_btree;
```

Hash Index

Hash indexes are used for equality comparisons. They are faster for equality queries, but do not support range queries.

Create a Hash index:

```
CREATE INDEX idx_name_hash ON employeepersonal USING HASH (lastname);
```

Query using a Hash index:

```
SELECT * FROM employeepersonal WHERE lastname = 'Smith';
```

Drop a Hash index:

```
DROP INDEX idx_name_hash;
```

GIN (Generalized Inverted Index)

GIN indexes are designed for indexing array, JSONB, and full-text search data.

Create a GIN index on an array:

```
CREATE INDEX idx_skills_gin ON employee_skills USING GIN (skills);
```

Query using a GIN index (array search):

```
SELECT * FROM employee_skills WHERE skills @> ARRAY['Python'];
```

Create a GIN index on a JSONB column:

```
CREATE INDEX idx_jsonb_gin ON employee_info USING GIN (details);
```

Full-text search with GIN index:

```
CREATE INDEX idx_fulltext_gin ON employeepersonal USING GIN (to_tsvector('english', firstname || ' ' || lastname));
```

Query full-text search using GIN index:

```
SELECT * FROM employeepersonal WHERE to_tsvector('english', firstname || ' ' || lastname) @@ to_tsquery('Doe');
```

GiST (Generalized Search Tree)

GiST indexes are used for more complex data types, like geometric data, ranges, or full-text search.

Create a GiST index:

```
CREATE INDEX idx_geom_gist ON spatial_table USING GIST (geom);
```

Range queries with GiST index:

```
SELECT * FROM reservations WHERE date_range && '[2023-09-01, 2023-09-10]':daterange;
```

Query using GiST index (spatial data):

```
SELECT * FROM spatial_table WHERE geom && 'BOX(1, 2, 3, 4)';
```

Create a GiST index on a composite column:

```
CREATE INDEX idx_date_gist ON reservations USING GIST (daterange);
```

Drop a GiST index:

```
DROP INDEX idx_geom_gist;
```

Creating and managing indexes

Indexes can significantly improve query performance, but they come with overhead in terms of disk space and maintenance during data inserts, updates, or deletions.

Create an Index:

```
CREATE INDEX idx_salary ON employeework (monthsalary);
```

Create a Partial Index (index only part of the table):

```
CREATE INDEX idx_active_employees ON employeework (monthsalary) WHERE monthsalary > 5000;
```

Create a Unique Index:

```
CREATE UNIQUE INDEX idx_unique_employee ON employeepersonal (email);
```

Create an Index Concurrently (to avoid locking):

```
CREATE INDEX CONCURRENTLY idx_email_concurrent ON employeepersonal (email);
```

Drop an Index:

```
DROP INDEX idx_salary;
```

Rename an Index:

```
ALTER INDEX idx_salary RENAME TO idx_employee_salary;
```

Reindex a table or index (rebuild a corrupted index):

```
REINDEX TABLE employeework;
```

List all indexes for a table:

```
SELECT indexname FROM pg_indexes WHERE tablename = 'employeework';
```

Create a Composite Index (multiple columns):

```
CREATE INDEX idx_composite ON employeework (department, position);
```

Disable an Index (for maintenance purposes):

```
ALTER INDEX idx_composite DISABLE;
```

Analyzing query performance with EXPLAIN

The EXPLAIN command shows how PostgreSQL executes a query, helping to analyze query performance by displaying the execution plan.

Basic EXPLAIN:

```
EXPLAIN SELECT * FROM employeepersonal WHERE lastname = 'Doe';
```

EXPLAIN with cost estimation:

```
EXPLAIN (COSTS TRUE) SELECT * FROM employeepersonal WHERE lastname = 'Smith';
```

EXPLAIN ANALYZE to show execution details:

```
EXPLAIN ANALYZE SELECT * FROM employeepersonal WHERE lastname = 'Doe';
```

Analyze a join query:

```
EXPLAIN SELECT ep.firstname, ew.department
FROM employeepersonal ep
JOIN employeework ew ON ep.id = ew.id WHERE ew.department = 'IT';
```

Analyze a query with an index:

```
EXPLAIN ANALYZE SELECT * FROM employeework WHERE monthsalary > 5000;
```

EXPLAIN a query with sorting:

```
EXPLAIN SELECT * FROM employeework ORDER BY monthsalary DESC;
```

EXPLAIN with a subquery:

```
EXPLAIN SELECT * FROM employeepersonal WHERE id IN (SELECT id FROM employeework WHERE department = 'IT');
```

EXPLAIN on an insert with a subquery:

```
EXPLAIN INSERT INTO employeework (position, department, monthsalary)
SELECT 'DevOps Engineer', 'IT', 6000 FROM employeepersonal WHERE lastname = 'Doe';
```

EXPLAIN a CTE query:

```
EXPLAIN WITH cte AS (
  SELECT department, AVG(monthsalary) AS avg_salary
  FROM employeework GROUP BY department
)
SELECT * FROM cte WHERE avg_salary > 5000;
```

Analyze a complex query:

```
EXPLAIN ANALYZE SELECT ep.firstname, ew.position
FROM employeepersonal ep
JOIN employeework ew ON ep.id = ew.id
WHERE ew.monthsalary > (SELECT AVG(monthsalary) FROM employeework);
```

Advanced Query Optimization

Understanding the PostgreSQL query planner

PostgreSQL's Query Planner determines the most efficient way to execute a query by considering factors such as indexes, table statistics, and query complexity.

Concepts:

- ✓ Sequential Scan: Scans the entire table row by row.
- ✓ Index Scan: Uses an index to fetch rows.
- ✓ Bitmap Index Scan: Uses a bitmap of matching rows from the index before fetching them.
- ✓ Join Methods:
 - Nested Loop: Joins two tables by scanning one and looking up each value in the other.
 - Hash Join: Builds a hash table for one side and probes it for matching rows.
 - Merge Join: Efficiently joins two pre-sorted tables.
- ✓ Parallel Execution: Uses multiple CPU cores to execute the query in parallel.

Analyzing query plans

Analyzing query plans helps you understand the PostgreSQL execution strategy and identify potential bottlenecks.

Sequential Scan:

```
EXPLAIN SELECT * FROM employeepersonal;
```

*The output will show a Seq Scan, indicating a full table scan.

Index Scan:

```
EXPLAIN SELECT * FROM employeepersonal WHERE lastname = 'Doe';
```

*The output will show an Index Scan if an index exists on lastname.

Bitmap Index Scan:

```
EXPLAIN SELECT * FROM employeepersonal WHERE firstname LIKE 'J%';
```

*Bitmap Index Scan is used for fetching multiple rows efficiently.

Nested Loop Join:

```
EXPLAIN SELECT ep.firstname, ew.department  
FROM employeepersonal ep  
JOIN employeework ew ON ep.id = ew.id;
```

*The query planner may use a Nested Loop Join.

Hash Join:

```
EXPLAIN SELECT ep.firstname, ew.department  
FROM employeepersonal ep  
JOIN employeework ew ON ep.id = ew.id WHERE ew.department = 'IT';
```

*If there's no index, a Hash Join might be used.

Merge Join:

```
EXPLAIN SELECT ep.firstname, ew.department
FROM employeepersonal ep
JOIN employeework ew ON ep.id = ew.id ORDER BY ew.department;
```

*A Merge Join may be used for ordered joins.

Parallel Execution:

```
EXPLAIN SELECT * FROM employeepersonal WHERE lastname = 'Doe';
```

*Parallel execution may be used if the table is large.

Techniques for query optimization

Optimizing queries helps reduce query execution time and resource usage.

Use indexes for faster lookups:

```
CREATE INDEX idx_lastname ON employeepersonal (lastname);
```

Avoid unnecessary SELECT * statements:

```
SELECT firstname, lastname FROM employeepersonal;
```

Use LIMIT for large result sets:

```
SELECT * FROM employeepersonal LIMIT 100;
```

Optimize JOIN queries with indexes:

```
CREATE INDEX idx_employee_id ON employeework (id);
```

Use appropriate data types:

```
ALTER TABLE employeework ALTER COLUMN monthsalary TYPE NUMERIC(10, 2);
```

Use partitions for large tables:

```
CREATE TABLE employeepartitioned PARTITION BY RANGE (datehired);
```

Optimize subqueries with JOIN:

```
SELECT ep.firstname, ew.department
FROM employeepersonal ep
JOIN employeework ew ON ep.id = ew.id;
```

Analyze and VACUUM regularly:

```
VACUUM ANALYZE employeepersonal;
```

Avoid large IN clauses:

```
SELECT * FROM employeepersonal WHERE id = ANY(ARRAY[1, 2, 3]);
```

Use CTEs for complex queries:

```
WITH high_salary AS (
  SELECT id, monthsalary FROM employeework WHERE monthsalary > 5000
)
SELECT * FROM high_salary;
```

Stored Procedures and Functions

Creating and using stored procedures

Stored procedures in PostgreSQL allow you to encapsulate a series of SQL statements in a reusable block of code. Unlike functions, stored procedures can perform operations like transactions, commit, and rollback, and don't necessarily return a value.

Basic Stored Procedure:

```
CREATE PROCEDURE insert_employee(IN firstname VARCHAR, IN lastname VARCHAR)
LANGUAGE SQL
AS $$
INSERT INTO employeepersonal (firstname, lastname)
VALUES (firstname, lastname);
$$;
```

Calling a Stored Procedure:

```
CALL insert_employee('John', 'Doe');
```

Stored Procedure with Output Parameters:

```
CREATE PROCEDURE get_employee(IN emp_id INT, OUT emp_name VARCHAR)
LANGUAGE SQL
AS $$
SELECT firstname || ' ' || lastname INTO emp_name FROM employeepersonal WHERE id = emp_id;
$$;
```

Stored Procedure with Transaction:

```
CREATE PROCEDURE transfer_funds(from_account INT, to_account INT, amount NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Begin transaction
    UPDATE accounts SET balance = balance - amount WHERE id = from_account;
    UPDATE accounts SET balance = balance + amount WHERE id = to_account;

    -- Check if the transfer is valid
    IF (SELECT balance FROM accounts WHERE id = from_account) < 0 THEN
        ROLLBACK;
    ELSE
        COMMIT;
    END IF;
END;
$$;
```

Stored Procedure with Conditional Logic:

```
CREATE PROCEDURE update_employee_salary(IN emp_id INT, IN new_salary NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    IF new_salary > 0 THEN
        UPDATE employeework SET monthllysalary = new_salary WHERE id = emp_id;
    ELSE
        RAISE EXCEPTION 'Invalid salary';
    END IF;
END;
$$;
```

Stored Procedure with Loop:

```
CREATE PROCEDURE loop_example()
LANGUAGE plpgsql
AS $$
DECLARE
    i INT;
BEGIN
    FOR i IN 1..10 LOOP
        INSERT INTO logs (message) VALUES ('Iteration ' || i);
    END LOOP;
END;
$$;
```

Stored Procedure with Dynamic SQL:

```
CREATE PROCEDURE dynamic_query(table_name VARCHAR)
LANGUAGE plpgsql
AS $$
BEGIN
    EXECUTE 'SELECT * FROM ' || quote_ident(table_name);
END;
$$;
```

Stored Procedure with Cursor:

```
CREATE PROCEDURE fetch_employees()
LANGUAGE plpgsql
AS $$
DECLARE
    emp_cursor CURSOR FOR SELECT firstname, lastname FROM employeepersonal;
    rec RECORD;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO rec;
        EXIT WHEN NOT FOUND;
        RAISE NOTICE 'Employee: % %', rec.firstname, rec.lastname;
    END LOOP;
```

```
CLOSE emp_cursor;
END;
$$;
```

Stored Procedure with Exception Handling:

```
CREATE PROCEDURE update_salary_with_exception(emp_id INT, new_salary NUMERIC)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE employeework SET monthllysalary = new_salary WHERE id = emp_id;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'An error occurred: division by zero';
    WHEN OTHERS THEN
        RAISE NOTICE 'An unexpected error occurred';
END;
$$;
```

Dropping a Stored Procedure:

```
DROP PROCEDURE IF EXISTS insert_employee;
```

Writing functions in PL/pgSQL

PL/pgSQL (Procedural Language/PostgreSQL) allows you to write functions that can be used as part of SQL queries. Functions in PL/pgSQL are reusable blocks of code that return a value and can perform more complex operations than standard SQL.

Basic Function Returning a Value:

```
CREATE FUNCTION get_fullname(emp_id INT) RETURNS VARCHAR
LANGUAGE plpgsql
AS $$
DECLARE
    fullname VARCHAR;
BEGIN
    SELECT firstname || ' ' || lastname INTO fullname FROM employeepersonal WHERE id = emp_id;
    RETURN fullname;
END;
$$;
```

Calling a Function:

```
SELECT get_fullname(1);
```

Function with Multiple Arguments:

```
CREATE FUNCTION calculate_bonus(emp_id INT, percentage NUMERIC) RETURNS NUMERIC
LANGUAGE plpgsql
AS $$
DECLARE
    salary NUMERIC;
BEGIN
```

```
SELECT monthsalary INTO salary FROM employeework WHERE id = emp_id;
RETURN salary * percentage / 100;
END;
$$;
```

Function with IF-THEN-ELSE Statement:

```
CREATE FUNCTION check_employee_status(emp_id INT) RETURNS VARCHAR
LANGUAGE plpgsql
AS $$
DECLARE
    hire_date DATE;
BEGIN
    SELECT datehired INTO hire_date FROM employeework WHERE id = emp_id;
    IF hire_date < NOW() - INTERVAL '1 year' THEN
        RETURN 'Permanent';
    ELSE
        RETURN 'Probation';
    END IF;
END;
$$;
```

Function with FOR Loop:

```
CREATE FUNCTION count_employees() RETURNS INT
LANGUAGE plpgsql
AS $$
DECLARE
    emp_count INT := 0;
BEGIN
    FOR emp IN SELECT * FROM employeepersonal LOOP
        emp_count := emp_count + 1;
    END LOOP;
    RETURN emp_count;
END;
$$;
```

Recursive Function Example:

```
CREATE FUNCTION factorial(n INT) RETURNS INT
LANGUAGE plpgsql
AS $$
BEGIN
    IF n = 0 THEN
        RETURN 1;
    ELSE
        RETURN n * factorial(n - 1);
    END IF;
END;
$$;
```

Function with RETURN QUERY:

```
CREATE FUNCTION get_high_salary_employees(salary_limit NUMERIC) RETURNS TABLE(firstname VARCHAR, lastname VARCHAR)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY SELECT firstname, lastname FROM employeepersonal
    JOIN employeework ON employeepersonal.id = employeework.id
    WHERE monthllysalary > salary_limit;
END;
$$;
```

Function with Exception Handling:

```
CREATE FUNCTION safe_division(numerator INT, denominator INT) RETURNS INT
LANGUAGE plpgsql
AS $$
BEGIN
    IF denominator = 0 THEN
        RAISE EXCEPTION 'Division by zero';
    ELSE
        RETURN numerator / denominator;
    END IF;
END;
$$;
```

Dropping a Function:

```
DROP FUNCTION IF EXISTS get_fullname(INT);
```

Function with Dynamic SQL:

```
CREATE FUNCTION dynamic_select(table_name VARCHAR, column_name VARCHAR) RETURNS SETOF RECORD
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY EXECUTE 'SELECT ' || column_name || ' FROM ' || table_name;
END;
$$;
```

Advanced use of PL/pgSQL features

PL/pgSQL offers advanced features for greater control, flexibility, and efficiency when writing stored functions and procedures. These include triggers, cursors, dynamic SQL, exception handling, and more.

Using Cursors:

```
CREATE FUNCTION fetch_employees_with_cursor() RETURNS VOID
LANGUAGE plpgsql
AS $$
DECLARE
    emp_cursor CURSOR FOR SELECT firstname, lastname FROM employeepersonal;
    rec RECORD;
BEGIN
```

```

OPEN emp_cursor;
LOOP
    FETCH emp_cursor INTO rec;
    EXIT WHEN NOT FOUND;
    RAISE NOTICE 'Employee: % %', rec.firstname, rec.lastname;
END LOOP;
CLOSE emp_cursor;
END;
$$;

```

Creating Triggers:

```

CREATE FUNCTION log_salary_change() RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF NEW.monthlysalary <> OLD.monthlysalary THEN
        INSERT INTO salary_logs (emp_id, old_salary, new_salary, change_date)
        VALUES (OLD.id, OLD.monthlysalary, NEW.monthlysalary, NOW());
    END IF;
    RETURN NEW;
END;
$$;

CREATE TRIGGER salary_update_trigger
AFTER UPDATE ON employeework
FOR EACH ROW
EXECUTE FUNCTION log_salary_change();

```

Using Dynamic SQL:

```

CREATE FUNCTION dynamic_update(table_name VARCHAR, column_name VARCHAR, new_value TEXT, condition TEXT)
RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN
    EXECUTE 'UPDATE ' || table_name || ' SET ' || column_name || ' = ' || quote_literal(new_value) || ' WHERE ' ||
condition;
END;
$$;

```

Function with Temporary Table:

```

CREATE FUNCTION create_temp_table() RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN
    CREATE TEMP TABLE temp_employees AS
    SELECT * FROM employeepersonal LIMIT 10;
END;
$$;

```

Function with Custom Exception Handling:

```
CREATE FUNCTION transfer_funds_safe(from_acc INT, to_acc INT, amount NUMERIC) RETURNS VOID
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE accounts SET balance = balance - amount WHERE id = from_acc;
    UPDATE accounts SET balance = balance + amount WHERE id = to_acc;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Division by zero error';
    WHEN OTHERS THEN
        RAISE NOTICE 'Unexpected error occurred';
END;
$$;
```

Function with Iterative Processing:

```
CREATE FUNCTION process_payroll() RETURNS VOID
LANGUAGE plpgsql
AS $$
DECLARE
    emp RECORD;
BEGIN
    FOR emp IN SELECT * FROM employeework LOOP
        UPDATE employeework SET monthllysalary = monthllysalary * 1.05 WHERE id = emp.id;
    END LOOP;
END;
$$;
```

Function with Record Type:

```
CREATE FUNCTION fetch_employee_details(emp_id INT) RETURNS RECORD
LANGUAGE plpgsql
AS $$
DECLARE
    emp RECORD;
BEGIN
    SELECT * INTO emp FROM employeepersonal WHERE id = emp_id;
    RETURN emp;
END;
$$;
```

Using EXCEPTION Clause for Error Handling:

```
CREATE FUNCTION divide_numbers(numerator INT, denominator INT) RETURNS INT
LANGUAGE plpgsql
AS $$
BEGIN
    IF denominator = 0 THEN
        RAISE EXCEPTION 'Division by zero error';
    END IF;
    RETURN numerator / denominator;
END;
```



```
END;  
$$;
```

Using RAISE NOTICE for Debugging:

```
CREATE FUNCTION calculate_tax(salary NUMERIC) RETURNS NUMERIC  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    RAISE NOTICE 'Calculating tax for salary: %', salary;  
    RETURN salary * 0.10;  
END;  
$$;
```

Dropping a Trigger:

```
DROP TRIGGER IF EXISTS salary_update_trigger ON employeework;
```

Triggers and Event-Driven Programming

Creating and managing triggers

Triggers in PostgreSQL are functions that are automatically executed or fired when specific database events (like INSERT, UPDATE, DELETE) occur on a table. They can be used for enforcing business rules, data integrity, or logging changes.

Creating a Trigger Function:

```
CREATE FUNCTION log_employee_update() RETURNS TRIGGER AS $$  
BEGIN  
    INSERT INTO employee_audit (emp_id, old_salary, new_salary, change_date)  
    VALUES (OLD.id, OLD.monthlysalary, NEW.monthlysalary, NOW());  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;
```

Create a Trigger:

```
CREATE TRIGGER trigger_employee_update  
AFTER UPDATE ON employeework  
FOR EACH ROW  
EXECUTE FUNCTION log_employee_update();
```

Trigger for INSERT:

```
CREATE TRIGGER trigger_employee_insert  
AFTER INSERT ON employeepersonal  
FOR EACH ROW  
EXECUTE FUNCTION log_employee_insert();
```

Trigger for DELETE:

```
CREATE TRIGGER trigger_employee_delete
AFTER DELETE ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION log_employee_delete();
```

Trigger for BEFORE UPDATE:

```
CREATE TRIGGER trigger_before_employee_update
BEFORE UPDATE ON employeework
FOR EACH ROW
EXECUTE FUNCTION before_update_function();
```

Disabling a Trigger:

```
ALTER TABLE employeework DISABLE TRIGGER trigger_employee_update;
```

Enabling a Trigger:

```
ALTER TABLE employeework ENABLE TRIGGER trigger_employee_update;
```

Dropping a Trigger:

```
DROP TRIGGER IF EXISTS trigger_employee_update ON employeework;
```

Listing Triggers on a Table:

```
SELECT tgname FROM pg_trigger WHERE tgrelid = 'employeework'::regclass;
```

Trigger for AFTER INSERT or UPDATE:

```
CREATE TRIGGER trigger_insert_or_update
AFTER INSERT OR UPDATE ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION insert_or_update_function();
```

Writing trigger functions

Trigger functions are special functions used by triggers. They are defined similarly to normal PostgreSQL functions, but they must return TRIGGER and are typically used to perform actions based on data changes.

Basic Trigger Function:

```
CREATE FUNCTION audit_log() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO audit_table (old_value, new_value, change_time)
    VALUES (OLD.monthlysalary, NEW.monthlysalary, NOW());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function for BEFORE INSERT:

```
CREATE FUNCTION before_insert_employee() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.firstname IS NULL THEN
        RAISE EXCEPTION 'First name cannot be null';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function for Data Validation:

```
CREATE FUNCTION validate_employee_age() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.birthdate > NOW() - INTERVAL '18 years' THEN
        RAISE EXCEPTION 'Employee must be at least 18 years old';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function to Prevent Deletion:

```
CREATE FUNCTION prevent_deletion() RETURNS TRIGGER AS $$
BEGIN
    IF OLD.position = 'Manager' THEN
        RAISE EXCEPTION 'Cannot delete a manager';
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function for AFTER DELETE:

```
CREATE FUNCTION log_employee_delete() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO employee_deletion_log (emp_id, deleted_at)
    VALUES (OLD.id, NOW());
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function for Data Modification:

```
CREATE FUNCTION increase_salary() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.position = 'Senior Engineer' THEN
        NEW.monthlysalary := NEW.monthlysalary * 1.10;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function to Log IP Address:

```
CREATE FUNCTION log_user_ip() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO user_log (user_id, action, ip_address, action_time)
    VALUES (NEW.id, TG_OP, inet_client_addr(), NOW());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function to Update Timestamp:

```
CREATE FUNCTION update_modified_time() RETURNS TRIGGER AS $$
BEGIN
    NEW.modified_at := NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function to Restrict Changes:

```
CREATE FUNCTION restrict_changes() RETURNS TRIGGER AS $$
BEGIN
    IF OLD.department <> NEW.department THEN
        RAISE EXCEPTION 'Cannot change department once set';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger Function for Audit Trail:

```
CREATE FUNCTION audit_employee_changes() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO employee_audit (emp_id, changes, changed_at)
    VALUES (OLD.id, hstore(OLD) - hstore(NEW), NOW());
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Using triggers for data validation and automation

Triggers are commonly used to enforce business rules, ensure data integrity, or automate routine tasks like logging, validation, or updating timestamps.

Trigger to Validate Salary:

```
CREATE FUNCTION validate_salary() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.monthlysalary < 0 THEN
        RAISE EXCEPTION 'Salary cannot be negative';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER validate_salary_trigger
BEFORE INSERT OR UPDATE ON employeework
FOR EACH ROW
EXECUTE FUNCTION validate_salary();
```

Trigger to Enforce Unique Combination:

```
CREATE FUNCTION enforce_unique_fullname() RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM employeepersonal WHERE firstname = NEW.firstname AND lastname = NEW.lastname)
    THEN
        RAISE EXCEPTION 'Employee with this full name already exists';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER unique_fullname_trigger
BEFORE INSERT ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION enforce_unique_fullname();
```

Trigger to Automatically Update Timestamp:

```
CREATE FUNCTION update_timestamp() RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at := NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_timestamp_trigger
BEFORE UPDATE ON employeework
FOR EACH ROW
EXECUTE FUNCTION update_timestamp();
```

Trigger for Auto-Generating an Employee Code:

```
CREATE FUNCTION generate_employee_code() RETURNS TRIGGER AS $$
BEGIN
    NEW.employee_code := 'EMP-' || NEW.id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER employee_code_trigger
BEFORE INSERT ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION generate_employee_code();
```

Trigger to Prevent Salary Reduction:

```
CREATE FUNCTION prevent_salary_reduction() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.monthlysalary < OLD.monthlysalary THEN
        RAISE EXCEPTION 'Salary cannot be reduced';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER salary_reduction_trigger
BEFORE UPDATE ON employeework
FOR EACH ROW
EXECUTE FUNCTION prevent_salary_reduction();
```

Trigger to Log Changes in Address:

```
CREATE FUNCTION log_address_change() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.address <> OLD.address THEN
        INSERT INTO address_change_log (emp_id, old_address, new_address, change_date)
        VALUES (OLD.id, OLD.address, NEW.address, NOW());
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER address_change_trigger
AFTER UPDATE ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION log_address_change();
```

This trigger checks whether the email being inserted or updated is in a valid format.

```
CREATE FUNCTION validate_email_format() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.email !~* '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$' THEN
        RAISE EXCEPTION 'Invalid email format: %', NEW.email;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER email_format_trigger
BEFORE INSERT OR UPDATE ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION validate_email_format();
```

This trigger updates the updated_at timestamp field whenever a row in the table is updated.

```
CREATE FUNCTION update_timestamp() RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at := NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_timestamp_trigger
BEFORE UPDATE ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION update_timestamp();
```

This trigger ensures that the employee is at least 18 years old.

```
CREATE FUNCTION validate_employee_age() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.birthdate > NOW() - INTERVAL '18 years' THEN
        RAISE EXCEPTION 'Employee must be at least 18 years old';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER employee_age_validation_trigger
BEFORE INSERT OR UPDATE ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION validate_employee_age();
```

If the department is not provided during an insert, this trigger sets a default department value.

```
CREATE FUNCTION set_default_department() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.department IS NULL THEN
        NEW.department := 'General';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER default_department_trigger
BEFORE INSERT ON employeeework
FOR EACH ROW
EXECUTE FUNCTION set_default_department();
```

This trigger prevents an employee's salary from being decreased.

```
CREATE FUNCTION prevent_salary_decrease() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.monthlysalary < OLD.monthlysalary THEN
        RAISE EXCEPTION 'Salary cannot be decreased';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER prevent_salary_decrease_trigger
BEFORE UPDATE ON employeeework
FOR EACH ROW
EXECUTE FUNCTION prevent_salary_decrease();
```

This trigger automatically generates an employee code based on the employee's ID.

```
CREATE FUNCTION generate_employee_code() RETURNS TRIGGER AS $$
BEGIN
    NEW.employee_code := 'EMP-' || NEW.id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER generate_employee_code_trigger
BEFORE INSERT ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION generate_employee_code();
```


This trigger logs any changes made to an employee's salary in a separate audit table.

```
CREATE FUNCTION log_salary_changes() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.monthlysalary <> OLD.monthlysalary THEN
        INSERT INTO salary_audit (emp_id, old_salary, new_salary, change_date)
        VALUES (OLD.id, OLD.monthlysalary, NEW.monthlysalary, NOW());
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER salary_change_trigger
AFTER UPDATE ON employeework
FOR EACH ROW
EXECUTE FUNCTION log_salary_changes();
```

This trigger validates the phone number format before inserting or updating.

```
CREATE FUNCTION validate_phone_number() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.phone !~ '^\\d{10}$' THEN
        RAISE EXCEPTION 'Phone number must be 10 digits';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER phone_number_validation_trigger
BEFORE INSERT OR UPDATE ON employeepersonal
FOR EACH ROW
EXECUTE FUNCTION validate_phone_number();
```

This trigger prevents the deletion of employees who have the position "Manager."

```
CREATE FUNCTION prevent_manager_deletion() RETURNS TRIGGER AS $$
BEGIN
    IF OLD.position = 'Manager' THEN
        RAISE EXCEPTION 'Cannot delete a Manager';
    END IF;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER prevent_manager_deletion_trigger
BEFORE DELETE ON employeework
FOR EACH ROW
EXECUTE FUNCTION prevent_manager_deletion();
```

This trigger automatically adjusts the employee rank based on salary changes.

```
CREATE FUNCTION adjust_employee_rank() RETURNS TRIGGER AS $$
BEGIN
    IF NEW.monthlysalary > 7000 THEN
        NEW.rank := 'Senior';
    ELSEIF NEW.monthlysalary > 5000 THEN
        NEW.rank := 'Mid-level';
    ELSE
        NEW.rank := 'Junior';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER employee_rank_trigger
BEFORE INSERT OR UPDATE ON employeework
FOR EACH ROW
EXECUTE FUNCTION adjust_employee_rank();
```

Transactions and Concurrency Control

Understanding transactions and ACID properties

A transaction in PostgreSQL is a sequence of database operations performed as a single logical unit of work.

Transactions must exhibit four key properties, known as ACID properties:

- **Atomicity:** Ensures that either all operations of a transaction are completed, or none are.
- **Consistency:** Guarantees that a transaction brings the database from one valid state to another, maintaining data integrity.
- **Isolation:** Ensures that transactions execute independently without interfering with each other.
- **Durability:** Ensures that once a transaction is committed, its changes are permanent, even in the event of a system failure.

Examples of Transactions and ACID Properties:

Basic Transaction:

```
BEGIN;
INSERT INTO employeepersonal (firstname, lastname) VALUES ('John', 'Doe');
COMMIT;
```

Atomicity Example (Ensuring all or nothing):

```
BEGIN;
UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
UPDATE accounts SET balance = balance + 1000 WHERE id = 2;
COMMIT;
```

Rollback Transaction (Fail Atomicity):

```
BEGIN;
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Jane', 'Doe');
ROLLBACK;
```

Consistency Example (Ensuring valid state transitions):

```
BEGIN;  
UPDATE employeepersonal SET firstname = NULL WHERE id = 1; -- Causes failure due to NOT NULL constraint  
COMMIT; -- Will not be executed due to violation
```

Isolation with Parallel Transactions:

```
-- Session 1  
BEGIN;  
UPDATE accounts SET balance = balance + 500 WHERE id = 1;  
  
-- Session 2 (attempts to access the same data)  
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
```

Durability Example:

```
BEGIN;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Tom', 'Smith');  
COMMIT; -- Ensures the data is saved permanently
```

Read Committed Isolation Level (Default isolation level in PostgreSQL):

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;  
BEGIN;  
SELECT * FROM employeepersonal WHERE id = 1;  
COMMIT;
```

Serializable Isolation Level:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
BEGIN;  
-- Ensures that transactions appear to execute in isolation  
SELECT * FROM employeepersonal WHERE id = 1;  
COMMIT;
```

Handling Errors in Transactions:

```
BEGIN;  
UPDATE accounts SET balance = balance - 500 WHERE id = 1;  
IF (SELECT balance FROM accounts WHERE id = 1) < 0 THEN  
    ROLLBACK;  
ELSE  
    COMMIT;  
END IF;
```

Multiple Operations in a Transaction:

```
BEGIN;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Alice', 'Cooper');  
UPDATE employeepersonal SET lastname = 'Smith' WHERE firstname = 'Alice';  
COMMIT;
```

Using SAVEPOINT and ROLLBACK

SAVEPOINT allows you to set intermediate points within a transaction. You can roll back to a specific savepoint without rolling back the entire transaction. ROLLBACK can be used to undo a transaction entirely or revert to a specific savepoint.

Examples of SAVEPOINT and ROLLBACK:

Creating a SAVEPOINT:

```
BEGIN;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('John', 'Doe');  
SAVEPOINT sp1;
```

Rolling Back to a SAVEPOINT:

```
ROLLBACK TO SAVEPOINT sp1;
```

COMMIT after SAVEPOINT:

```
COMMIT;
```

SAVEPOINT with Conditional ROLLBACK:

```
BEGIN;  
SAVEPOINT sp1;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Jane', 'Doe');  
IF (SELECT COUNT(*) FROM employeepersonal) > 10 THEN  
    ROLLBACK TO SAVEPOINT sp1;  
END IF;  
COMMIT;
```

SAVEPOINT and Nested Transactions:

```
BEGIN;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Tom', 'Smith');  
SAVEPOINT sp1;  
UPDATE employeepersonal SET lastname = 'Johnson' WHERE firstname = 'Tom';  
ROLLBACK TO SAVEPOINT sp1; -- Rolls back only the update  
COMMIT;
```

Using SAVEPOINT with Error Handling:

```
BEGIN;  
SAVEPOINT sp1;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Alice', 'Walker');  
EXCEPTION WHEN OTHERS THEN  
    ROLLBACK TO SAVEPOINT sp1;  
END;  
COMMIT;
```

Multiple SAVEPOINTS in a Transaction:

```
BEGIN;  
SAVEPOINT sp1;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('John', 'Doe');  
SAVEPOINT sp2;  
UPDATE employeepersonal SET lastname = 'Smith' WHERE firstname = 'John';  
ROLLBACK TO SAVEPOINT sp1; -- Rolls back to sp1, discards sp2  
COMMIT;
```

Dropping a SAVEPOINT:

```
RELEASE SAVEPOINT sp1;
```

Handling Multiple Savepoints with ROLLBACK:

```
BEGIN;  
SAVEPOINT sp1;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Jane', 'Doe');  
SAVEPOINT sp2;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('John', 'Smith');  
ROLLBACK TO SAVEPOINT sp1; -- Discards both sp1 and sp2  
COMMIT;
```

COMMIT without Rolling Back Savepoints:

```
BEGIN;  
SAVEPOINT sp1;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('Alice', 'Cooper');  
SAVEPOINT sp2;  
INSERT INTO employeepersonal (firstname, lastname) VALUES ('John', 'Doe');  
COMMIT; -- All changes are committed, even after creating savepoints
```

Managing concurrency with locking mechanisms

PostgreSQL uses locking mechanisms to manage concurrent access to data, ensuring data integrity in multi-user environments. There are several types of locks:

- Table-level locks: Lock an entire table.
- Row-level locks: Lock specific rows.
- Advisory locks: User-defined locks for custom concurrency control.

Examples of Locking Mechanisms:

Table-level Lock (for exclusive write access):

```
LOCK TABLE employeepersonal IN ACCESS EXCLUSIVE MODE;
```

Row-level Lock (using FOR UPDATE):

```
SELECT * FROM employeepersonal WHERE id = 1 FOR UPDATE;
```

Row-level Lock (FOR SHARE):

```
SELECT * FROM employeepersonal WHERE id = 1 FOR SHARE;
```

Lock with NOWAIT (avoiding waiting for a lock):

```
SELECT * FROM employeepersonal WHERE id = 1 FOR UPDATE NOWAIT;
```

Advisory Lock (user-defined locking):

```
SELECT pg_advisory_lock(12345);
```

Release Advisory Lock:

```
SELECT pg_advisory_unlock(12345);
```

Advisory Lock on Rows:

```
SELECT pg_advisory_lock(id) FROM employeepersonal WHERE id = 1;
```

Using SKIP LOCKED (skip locked rows):

```
SELECT * FROM employeepersonal WHERE id = 1 FOR UPDATE SKIP LOCKED;
```

Detect Deadlocks:

```
-- Session 1
BEGIN;
SELECT * FROM employeepersonal WHERE id = 1 FOR UPDATE;
-- Session 2 (Deadlock occurs when trying to access the same row)
SELECT * FROM employeepersonal WHERE id = 1 FOR UPDATE;
```

Check for Deadlocks with pg_locks:

```
SELECT * FROM pg_locks WHERE relation::regclass = 'employeepersonal'::regclass;
```

Security and User Management

Role-based access control

Role-based access control (RBAC) is a method of restricting system access to authorized users based on roles.

PostgreSQL implements RBAC using roles, which can be thought of as users or groups with a specific set of privileges.

Roles can be granted different levels of permissions (SELECT, INSERT, UPDATE, DELETE, etc.) on database objects such as tables, schemas, and databases.

Key Points in RBAC:

- ✓ Roles can be assigned to other roles or users.
- ✓ Privileges define what actions can be performed.
- ✓ Access to database objects (tables, views, functions) is controlled by granting or revoking privileges.

Examples of Role-based Access Control:

Create a Role:

```
CREATE ROLE readonly;
```

Create a User and Assign a Role:

```
CREATE USER john WITH PASSWORD 'password123';
GRANT readonly TO john;
```

Grant SELECT Privileges to a Role:

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;
```

Grant Privileges on Specific Table to a Role:

```
GRANT SELECT, INSERT ON employees TO readonly;
```

Grant EXECUTE on Functions:

```
GRANT EXECUTE ON FUNCTION calculate_salary() TO readonly;
```

Create a Role with LOGIN Privilege:

```
CREATE ROLE manager WITH LOGIN PASSWORD 'managerpass';
```

Grant ALL Privileges to a Role on a Database:

```
GRANT ALL PRIVILEGES ON DATABASE company_db TO manager;
```

Revoke Privileges from a Role:

```
REVOKE INSERT, UPDATE ON employees FROM readonly;
```

Check Role Privileges:

```
\du john
```

Drop a Role:

```
DROP ROLE readonly;
```

Managing permissions and roles

In PostgreSQL, permissions are managed at the object level (tables, views, sequences, etc.). Permissions are granted or revoked based on roles, and each role can have its own set of privileges. Effective management of roles and permissions ensures that users have the appropriate level of access and no more than is necessary.

Key Concepts:

- ✓ GRANT: Assign privileges to roles.
- ✓ REVOKE: Remove privileges from roles.
- ✓ INHERIT: Determines whether a role inherits the privileges of the roles it is a member of.
- ✓ SUPERUSER: A role with unrestricted access and control over the database.

Examples of Managing Permissions and Roles:

Grant SELECT Privileges to a Role on a Schema:

```
GRANT SELECT ON ALL TABLES IN SCHEMA public TO data_reader;
```

Grant UPDATE Privileges on a Specific Column:

```
GRANT UPDATE (salary) ON employees TO manager;
```

Grant Role to Another Role (Role Hierarchy):

```
GRANT data_reader TO junior_staff;
```

Grant ALL Privileges on a Table to a Role:

```
GRANT ALL PRIVILEGES ON TABLE employees TO admin;
```

Revoke DELETE Privileges from a Role:

```
REVOKE DELETE ON employees FROM readonly;
```

Revoke ALL Privileges on a Table:

```
REVOKE ALL PRIVILEGES ON employees FROM manager;
```

Alter a Role to Grant SUPERUSER Status:

```
ALTER ROLE admin WITH SUPERUSER;
```

Create a Role without LOGIN Privilege:

```
CREATE ROLE analyst NOLOGIN;
```

Grant Privileges on a Sequence:

```
GRANT USAGE, SELECT ON SEQUENCE employee_id_seq TO analyst;
```

Change the Password of a User Role:

```
ALTER ROLE john WITH PASSWORD 'newpassword123';
```

Best practices for database security

Database security is a critical part of managing databases. Implementing best practices ensures that data is protected from unauthorized access, tampering, and loss. PostgreSQL provides various security mechanisms to ensure the confidentiality, integrity, and availability of the data.

Use Role-based Access Control (RBAC):

Always use roles to assign permissions instead of granting privileges to individual users. This makes privilege management easier.

```
CREATE ROLE read_only;  
GRANT SELECT ON ALL TABLES IN SCHEMA public TO read_only;
```

Enforce Password Policies:

Ensure users have strong passwords and update them regularly.

```
ALTER ROLE john WITH PASSWORD 'ComplexPass123!';
```

Use SSL/TLS for Secure Connections:

Enable SSL/TLS to encrypt data in transit between the client and the server. In postgresql.conf:

```
ssl = on
```

Grant the Minimum Privileges Necessary:

Follow the principle of least privilege. Only grant the permissions that are absolutely necessary for users.

```
GRANT SELECT ON employees TO readonly;
```

Use NOINHERIT for Sensitive Roles:

Use NOINHERIT to prevent a user from automatically inheriting privileges from roles they're part of.

```
CREATE ROLE admin NOINHERIT;
```


Enable Logging for Security Monitoring:

Enable and configure logging for auditing and monitoring unauthorized access attempts. In postgresql.conf:

```
log_connections = on  
log_disconnections = on
```

Restrict Superuser and Database Owner Access:

Limit the number of users with SUPERUSER privileges and ensure they're trusted.

```
REVOKE SUPERUSER FROM admin;
```

Use Network-Level Security:

Use PostgreSQL's pg_hba.conf file to restrict who can connect to your database. In pg_hba.conf:

```
host all all 192.168.1.0/24 md5
```

Use Strong Authentication Mechanisms:

Use strong authentication mechanisms like SCRAM-SHA-256. In postgresql.conf:

```
password_encryption = 'scram-sha-256'
```

Encrypt Sensitive Data at Rest:

Use encryption mechanisms for sensitive data, especially for backups and archived data. In postgresql.conf:

```
data_encryption = 'on'
```

Introduction to Analytics with PostgreSQL

Overview of analytical capabilities in PostgreSQL

PostgreSQL is not only a transactional database, but it also provides a wide range of analytical capabilities. These features allow you to perform complex data analysis, aggregation, and reporting on large datasets. Key features for analytics include window functions, aggregations, grouping sets, CTEs, partitioning, and parallel query execution.

Key Analytical Features:

- ✓ Window Functions: Perform calculations across a set of table rows related to the current row.
- ✓ Aggregations: Compute statistics like SUM, AVG, COUNT, etc.
- ✓ CTEs (Common Table Expressions): Break down complex queries for readability and reusability.
- ✓ Grouping Sets: Perform multiple levels of aggregation in one query.
- ✓ Partitioning: Divide large tables for better query performance.
- ✓ Parallel Query Execution: Utilize multiple CPU cores to improve query performance on large datasets.

Examples of Analytical Capabilities in PostgreSQL:

Using Window Functions (e.g., ROW_NUMBER):

```
SELECT id, firstname, lastname,  
       ROW_NUMBER() OVER (ORDER BY monthsalary DESC) AS rank  
FROM employeework;
```

Aggregate Function (SUM):

```
SELECT department, SUM(monthsalary) AS total_salary  
FROM employeework  
GROUP BY department;
```

Using CTEs for Data Transformation:

```
WITH high_salary_employees AS (  
    SELECT id, firstname, lastname, monthsalary  
    FROM employeeework  
    WHERE monthsalary > 5000  
)  
SELECT * FROM high_salary_employees;
```

GROUPING SETS for Advanced Aggregation:

```
SELECT department, position, SUM(monthsalary)  
FROM employeeework  
GROUP BY GROUPING SETS (  
    (department),  
    (department, position)  
);
```

Using PARTITION BY for Advanced Analytics:

```
SELECT id, firstname, lastname, monthsalary,  
    AVG(monthsalary) OVER (PARTITION BY department) AS avg_salary  
FROM employeeework;
```

Parallel Query Execution:

```
SET max_parallel_workers_per_gather = 4;  
SELECT COUNT(*)  
FROM employeeework  
WHERE monthsalary > 5000;
```

LAG Function for Time-based Analysis:

```
SELECT id, firstname, lastname, monthsalary,  
    LAG(monthsalary, 1) OVER (ORDER BY monthsalary DESC) AS prev_salary  
FROM employeeework;
```

CUME_DIST for Percentile Analysis:

```
SELECT id, firstname, monthsalary,  
    CUME_DIST() OVER (ORDER BY monthsalary DESC) AS percentile_rank  
FROM employeeework;
```

Rank Employees by Salary:

```
SELECT id, firstname, monthsalary,  
    RANK() OVER (ORDER BY monthsalary DESC) AS salary_rank  
FROM employeeework;
```

ROLLUP for Hierarchical Aggregation:

```
SELECT department, position, SUM(monthsalary)  
FROM employeeework  
GROUP BY ROLLUP (department, position);
```

Setting up a data warehouse environment

A data warehouse is a system used for reporting and data analysis, typically consisting of large volumes of data. PostgreSQL can be used as a data warehouse for analytical workloads, supporting features like partitioning, indexing, and parallel processing for high-performance queries.

Steps to Set Up a Data Warehouse:

- ✓ Schema Design: Organize your data into fact and dimension tables.
- ✓ Partitioning: Divide large tables for performance.
- ✓ Indexing: Use indexes to speed up query performance.
- ✓ ETL (Extract, Transform, Load): Load data into the data warehouse.
- ✓ Materialized Views: Precompute and store query results.

Examples of Setting Up a Data Warehouse:

Creating a Fact Table:

```
CREATE TABLE sales (  
    sale_id SERIAL PRIMARY KEY,  
    customer_id INT,  
    product_id INT,  
    sale_date DATE,  
    amount NUMERIC  
);
```

Creating a Dimension Table:

```
CREATE TABLE customers (  
    customer_id SERIAL PRIMARY KEY,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    city VARCHAR(50)  
);
```

Creating a Partitioned Table:

```
CREATE TABLE sales_2023 (  
    LIKE sales INCLUDING ALL  
    ) PARTITION BY RANGE (sale_date);
```

Creating Indexes for Performance:

```
CREATE INDEX idx_sales_date ON sales (sale_date);  
CREATE INDEX idx_sales_customer_id ON sales (customer_id);
```

Setting Up Foreign Key Relationships:

```
ALTER TABLE sales  
ADD CONSTRAINT fk_customer  
FOREIGN KEY (customer_id) REFERENCES customers (customer_id);
```

Using Materialized Views:

```
CREATE MATERIALIZED VIEW total_sales_by_product AS  
SELECT product_id, SUM(amount) AS total_sales  
FROM sales  
GROUP BY product_id;
```

Creating a Star Schema:

```
-- Fact Table: Sales
-- Dimension Table: Time, Customer, Product
CREATE TABLE time_dim (
    time_id SERIAL PRIMARY KEY,
    sale_date DATE,
    month INT,
    year INT
);
```

Partition a Large Table by Date:

```
CREATE TABLE sales_partitioned
PARTITION BY RANGE (sale_date);

CREATE TABLE sales_q1 PARTITION OF sales_partitioned
FOR VALUES FROM ('2023-01-01') TO ('2023-03-31');
```

Use Foreign Data Wrappers (FDW) to Connect to External Data Sources:

```
CREATE EXTENSION postgres_fdw;
CREATE SERVER external_db_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'remote_host', dbname 'external_db', port '5432');
```

Pre-aggregating Data with Materialized Views for Faster Queries:

```
REFRESH MATERIALIZED VIEW total_sales_by_product;
```

ETL processes using PostgreSQL

ETL (Extract, Transform, Load) processes are used to extract data from multiple sources, transform it into a usable format, and load it into a data warehouse or database for analysis. PostgreSQL supports powerful SQL features and extensions like FDW (Foreign Data Wrappers) and PL/pgSQL for ETL workflows.

Key Steps in ETL:

- ✓ Extract: Retrieve data from multiple sources (databases, files, APIs).
- ✓ Transform: Clean, aggregate, or reshape the data.
- ✓ Load: Insert transformed data into the target database.

Examples of ETL Processes Using PostgreSQL:

Extract Data from a CSV File:

```
COPY employees FROM '/path/to/employees.csv' DELIMITER ',' CSV HEADER;
```

Extract Data from an External Database:

```
CREATE EXTENSION postgres_fdw;
CREATE SERVER ext_server FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host '192.168.1.100', dbname 'source_db');
```

Transform Data (Clean Null Values):

```
UPDATE employees
SET email = 'unknown@example.com'
WHERE email IS NULL;
```

Aggregate Data for Transformation:

```
INSERT INTO sales_aggregated (product_id, total_sales, year)
SELECT product_id, SUM(amount), EXTRACT(YEAR FROM sale_date)
FROM sales
GROUP BY product_id, EXTRACT(YEAR FROM sale_date);
```

Load Data into a Warehouse Table:

```
INSERT INTO sales_warehouse (product_id, customer_id, sale_date, amount)
SELECT product_id, customer_id, sale_date, amount
FROM sales_staging;
```

Use CTE for Transformation:

```
WITH clean_data AS (
  SELECT id, firstname, lastname, TRIM(email) AS email
  FROM employees_staging
)
INSERT INTO employees (id, firstname, lastname, email)
SELECT * FROM clean_data;
```

Transform Data by Splitting Fields:

```
INSERT INTO employee_names (firstname, lastname)
SELECT SPLIT_PART(fullname, ' ', 1), SPLIT_PART(fullname, ' ', 2)
FROM employees_staging;
```

Using FDW to Extract and Transform Data from External Databases:

```
CREATE FOREIGN TABLE external_sales (
  sale_id INT,
  sale_date DATE,
  amount NUMERIC
) SERVER ext_server OPTIONS (schema_name 'public', table_name 'sales');
```

Trigger-based ETL Process for Loading Data:

```
CREATE TRIGGER after_insert_sales
AFTER INSERT ON sales_staging
FOR EACH ROW
EXECUTE FUNCTION transform_and_load_sales();
```

Scheduled ETL Process Using pgAgent:

```
-- Use pgAgent or cron jobs to schedule regular ETL processes
INSERT INTO sales_warehouse (product_id, sale_date, total_sales)
SELECT product_id, sale_date, SUM(amount)
FROM sales_staging
WHERE sale_date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY product_id, sale_date;
```

Advanced Analytical Queries

Using window functions for analytics

Window functions are used to perform calculations across a set of rows related to the current row. Unlike aggregate functions, window functions do not collapse rows into a single output; instead, they allow you to work with individual rows while still considering the entire dataset.

Key Window Functions:

- ROW_NUMBER(): Assigns a unique sequential integer to rows within a partition.
- RANK(): Assigns a rank to rows, with gaps in ranking for ties.
- DENSE_RANK(): Similar to RANK but without gaps for ties.
- LEAD() / LAG(): Accesses the next or previous row in a dataset.
- FIRST_VALUE() / LAST_VALUE(): Returns the first or last value in a partition.

Examples of Window Functions:

ROW_NUMBER():

```
SELECT id, firstname, lastname,
       ROW_NUMBER() OVER (ORDER BY monthsalary DESC) AS row_num
FROM employeework;
```

RANK():

```
SELECT id, firstname, monthsalary,
       RANK() OVER (ORDER BY monthsalary DESC) AS salary_rank
FROM employeework;
```

DENSE_RANK():

```
SELECT id, firstname, monthsalary,
       DENSE_RANK() OVER (ORDER BY monthsalary DESC) AS dense_rank
FROM employeework;
```

LEAD():

```
SELECT id, firstname, monthsalary,
       LEAD(monthsalary, 1) OVER (ORDER BY monthsalary DESC) AS next_salary
FROM employeework;
```

LAG():

```
SELECT id, firstname, monthsalary,
       LAG(monthsalary, 1) OVER (ORDER BY monthsalary DESC) AS prev_salary
FROM employeework;
```

FIRST_VALUE():

```
SELECT id, firstname, monthsalary,  
       FIRST_VALUE(monthsalary) OVER (PARTITION BY department ORDER BY monthsalary DESC) AS highest_salary  
FROM employeework;
```

LAST_VALUE():

```
SELECT id, firstname, monthsalary,  
       LAST_VALUE(monthsalary) OVER (PARTITION BY department ORDER BY monthsalary DESC) AS lowest_salary  
FROM employeework;
```

NTILE():

```
SELECT id, firstname, monthsalary,  
       NTILE(4) OVER (ORDER BY monthsalary DESC) AS salary_quartile  
FROM employeework;
```

CUME_DIST():

```
SELECT id, firstname, monthsalary,  
       CUME_DIST() OVER (ORDER BY monthsalary DESC) AS cumulative_dist  
FROM employeework;
```

PERCENT_RANK():

```
SELECT id, firstname, monthsalary,  
       PERCENT_RANK() OVER (ORDER BY monthsalary DESC) AS percent_rank  
FROM employeework;
```

Aggregate functions and GROUP BY extensions

Aggregate functions in PostgreSQL are used to compute a single result from a set of input values. These functions are often used with GROUP BY to aggregate data based on specific columns. PostgreSQL also provides extensions to GROUP BY like GROUPING SETS, ROLLUP, and CUBE for multi-level aggregation.

Key Aggregate Functions:

- SUM(): Calculates the total sum of a column.
- AVG(): Computes the average of a set of values.
- COUNT(): Counts the number of rows.
- MAX() / MIN(): Returns the maximum or minimum value in a column.
- GROUPING SETS, ROLLUP, CUBE: Perform advanced multi-level aggregation.

Examples of Aggregate Functions and GROUP BY Extensions:

SUM():

```
SELECT department, SUM(monthsalary) AS total_salary  
FROM employeework  
GROUP BY department;
```

AVG():

```
SELECT department, AVG(monthlysalary) AS avg_salary
FROM employeework
GROUP BY department;
```

COUNT():

```
SELECT department, COUNT(*) AS employee_count
FROM employeework
GROUP BY department;
```

MAX():

```
SELECT department, MAX(monthlysalary) AS highest_salary
FROM employeework
GROUP BY department;
```

MIN():

```
SELECT department, MIN(monthlysalary) AS lowest_salary
FROM employeework
GROUP BY department;
```

GROUPING SETS:

```
SELECT department, position, SUM(monthlysalary)
FROM employeework
GROUP BY GROUPING SETS (
    (department),
    (department, position)
);
```

ROLLUP:

```
SELECT department, position, SUM(monthlysalary)
FROM employeework
GROUP BY ROLLUP (department, position);
```

CUBE:

```
SELECT department, position, SUM(monthlysalary)
FROM employeework
GROUP BY CUBE (department, position);
```

HAVING Clause for Filtering Aggregates:

```
SELECT department, SUM(monthlysalary) AS total_salary
FROM employeework
GROUP BY department
HAVING SUM(monthlysalary) > 20000;
```


GROUPING and Aggregate Functions:

```
SELECT department, SUM(monthlysalary),  
    GROUPING(department) AS dept_grouping  
FROM employeework  
GROUP BY ROLLUP (department);
```

Pivoting and unpivoting data

Pivoting refers to converting rows into columns, which is useful for transforming data for reports and analysis.

Unpivoting is the reverse process, converting columns into rows. PostgreSQL doesn't have built-in PIVOT or UNPIVOT functionality like some databases, but you can achieve this using CASE expressions, CROSS JOIN, and GROUP BY.

Examples of Pivoting and Unpivoting Data:

Basic Pivot with CASE:

```
SELECT department,  
    SUM(CASE WHEN position = 'Manager' THEN monthlysalary ELSE 0 END) AS manager_salary,  
    SUM(CASE WHEN position = 'Engineer' THEN monthlysalary ELSE 0 END) AS engineer_salary  
FROM employeework  
GROUP BY department;
```

Dynamic Pivot using GROUP BY and FILTER:

```
SELECT department,  
    SUM(monthlysalary) FILTER (WHERE position = 'Manager') AS manager_salary,  
    SUM(monthlysalary) FILTER (WHERE position = 'Engineer') AS engineer_salary  
FROM employeework  
GROUP BY department;
```

Pivoting Multiple Columns:

```
SELECT department,  
    COUNT(CASE WHEN position = 'Manager' THEN 1 END) AS manager_count,  
    COUNT(CASE WHEN position = 'Engineer' THEN 1 END) AS engineer_count  
FROM employeework  
GROUP BY department;
```

Pivot with Aggregate Functions:

```
SELECT department,  
    MAX(CASE WHEN position = 'Manager' THEN monthlysalary ELSE 0 END) AS max_manager_salary,  
    MAX(CASE WHEN position = 'Engineer' THEN monthlysalary ELSE 0 END) AS max_engineer_salary  
FROM employeework  
GROUP BY department;
```

Unpivot Using CROSS JOIN:

```
SELECT emp_id, attribute, value  
FROM employees e  
CROSS JOIN LATERAL (VALUES  
    ('firstname', firstname),  
    ('lastname', lastname),  
    ('email', email)  
) AS unpivoted(attribute, value);
```

Unpivot Specific Columns:

```
SELECT id, 'monthsalary' AS attribute, monthsalary AS value
FROM employeeework
UNION ALL
SELECT id, 'bonus', bonus FROM employeeework;
```

Pivot Multiple Conditions:

```
SELECT department,
       SUM(CASE WHEN position = 'Manager' AND monthsalary > 5000 THEN 1 ELSE 0 END) AS high_paid_managers,
       SUM(CASE WHEN position = 'Engineer' AND monthsalary > 5000 THEN 1 ELSE 0 END) AS high_paid_engineers
FROM employeeework
GROUP BY department;
```

Unpivot with Multiple Columns:

```
SELECT id, attribute, value
FROM employeeework
CROSS JOIN LATERAL (VALUES
  ('monthsalary', monthsalary::TEXT),
  ('bonus', bonus::TEXT)
) AS unpivoted(attribute, value);
```

Conditional Pivoting with CASE:

```
SELECT department,
       COUNT(CASE WHEN position = 'Manager' THEN 1 END) AS manager_count,
       COUNT(CASE WHEN position = 'Engineer' THEN 1 END) AS engineer_count
FROM employeeework
GROUP BY department;
```

Complex Pivot with Multiple Aggregates:

```
SELECT department,
       AVG(CASE WHEN position = 'Manager' THEN monthsalary ELSE NULL END) AS avg_manager_salary,
       AVG(CASE WHEN position = 'Engineer' THEN monthsalary ELSE NULL END) AS avg_engineer_salary
FROM employeeework
GROUP BY department;
```

Data Visualization and Reporting

Integrating PostgreSQL with BI tools (Microsoft Power BI)

Power BI is a popular Business Intelligence (BI) tool that allows users to connect to various data sources, including PostgreSQL, to create interactive reports and visualizations. PostgreSQL integration with Power BI can be done using ODBC drivers or native connectors provided by Power BI.

Integrating PostgreSQL with Power BI:

1. Install PostgreSQL ODBC Driver:

Download the PostgreSQL ODBC driver (psqlODBC) from the official PostgreSQL website and install it on the machine running Power BI. <https://odbc.postgresql.org/>

2. Connect Power BI to PostgreSQL:

Open Power BI, click on Get Data, select PostgreSQL database, and enter the server name, database name, and authentication details.

Example: Server=localhost; Database=mydb;

3. Import a Table into Power BI:

Once connected, select the table(s) from PostgreSQL you want to import and click Load.

Example: Import sales and customers tables from the database.

4. Write SQL Queries in Power BI to Fetch Data:

Instead of loading tables, you can use SQL queries to fetch specific data.

```
SELECT customer_id, firstname, lastname, total_spent
FROM customers
WHERE total_spent > 1000;
```

5. Use Power Query Editor to Transform Data:

Clean and transform the data within Power BI using Power Query Editor (e.g., remove nulls, change data types).

6. Create a Visualization in Power BI:

After loading data, create a visualization (e.g., a bar chart) that shows total sales by customer using data from PostgreSQL.

7. Filter Data for a Specific Time Range:

In Power BI, apply a time filter to display data for a specific period, such as "Sales in Q1 2023."

8. Create a Calculated Column in Power BI:

Create a calculated column in Power BI based on data from PostgreSQL. For example, calculate the customer age based on the birthdate.

```
Age = DATEDIFF([birthdate], TODAY(), YEAR)
```

9. Use Relationships in Power BI:

Define relationships between tables (e.g., sales and customers) based on foreign keys.

10. Publish the Power BI Report to Power BI Service:

After building the report, publish it to Power BI Service so other users can view it.

Creating views for reporting

Views in PostgreSQL are virtual tables created by a query. They are used to simplify complex queries, hide data complexity, or provide a specific subset of data for reporting purposes. Views are commonly used for reporting because they allow for easier data access and improved readability.

Benefits of Views:

- ✓ Simplify Complex Queries: A view can encapsulate complex SQL logic.
- ✓ Security: Views can restrict access to sensitive data.
- ✓ Reusability: Views can be reused across multiple reports or queries.

Examples of Creating Views for Reporting:

Create a Simple View:

```
CREATE VIEW view_sales_summary AS
SELECT product_id, SUM(amount) AS total_sales
FROM sales
GROUP BY product_id;
```

Create a View with Join:

```
CREATE VIEW view_customer_sales AS
SELECT c.customer_id, c.firstname, c.lastname, SUM(s.amount) AS total_sales
FROM customers c
JOIN sales s ON c.customer_id = s.customer_id
GROUP BY c.customer_id;
```

Create a View with Aggregation:

```
CREATE VIEW view_department_salaries AS
SELECT department, AVG(monthlysalary) AS avg_salary, SUM(monthlysalary) AS total_salary
FROM employeework
GROUP BY department;
```

Create a View with Filtering:

```
CREATE VIEW view_high_value_customers AS
SELECT customer_id, firstname, lastname, total_spent
FROM customers
WHERE total_spent > 10000;
```

Create a View for Reporting Based on Time Range:

```
CREATE VIEW view_sales_last_year AS
SELECT product_id, SUM(amount) AS total_sales
FROM sales
WHERE sale_date BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY product_id;
```

Create a View with Conditional Logic:

```
CREATE VIEW view_employee_bonus AS
SELECT id, firstname, lastname,
CASE
    WHEN monthlysalary > 5000 THEN 'Eligible'
```

```
ELSE 'Not Eligible'
END AS bonus_eligibility
FROM employeeework;
```

Create a View for Customer Segmentation:

```
CREATE VIEW view_customer_segments AS
SELECT customer_id, firstname, lastname,
CASE
    WHEN total_spent > 10000 THEN 'Premium'
    WHEN total_spent BETWEEN 5000 AND 10000 THEN 'Gold'
    ELSE 'Regular'
END AS segment
FROM customers;
```

Create a View with Subquery:

```
CREATE VIEW view_employee_ranking AS
SELECT id, firstname, lastname, monthllysalary,
(SELECT COUNT(*) FROM employeeework WHERE monthllysalary > e.monthllysalary) + 1 AS salary_rank
FROM employeeework e;
```

Create a Materialized View for Reporting:

Materialized views store the query result on disk, which can be refreshed periodically for faster access.

```
CREATE MATERIALIZED VIEW mat_view_sales_summary AS
SELECT product_id, SUM(amount) AS total_sales
FROM sales
GROUP BY product_id;
```

Drop a View:

```
DROP VIEW IF EXISTS view_employee_ranking;
```

Using PostgreSQL with Python for data analysis and visualization

Python is widely used for data analysis and visualization. Using libraries like psycopg2 or SQLAlchemy, you can easily connect to PostgreSQL databases, run queries, and analyze the data using Python's powerful libraries such as Pandas, Matplotlib, and Seaborn.

Key Python Libraries:

- ✓ psycopg2: A PostgreSQL database adapter for Python.
- ✓ SQLAlchemy: A SQL toolkit and Object-Relational Mapping (ORM) library.
- ✓ Pandas: For data manipulation and analysis.
- ✓ Matplotlib and Seaborn: For data visualization.

Examples of Using PostgreSQL with Python for Data Analysis and Visualization:

Connect to PostgreSQL Using psycopg2:

[python]

```
import psycopg2
conn = psycopg2.connect(
    dbname="mydb", user="myuser", password="mypassword", host="localhost"
)
cursor = conn.cursor()
```

Run a Query and Fetch Data:

[python]

```
cursor.execute("SELECT * FROM sales LIMIT 10")
results = cursor.fetchall()
for row in results:
    print(row)
```

Load Data into Pandas DataFrame:

[python]

```
import pandas as pd
query = "SELECT product_id, SUM(amount) AS total_sales FROM sales GROUP BY product_id"
df = pd.read_sql_query(query, conn)
print(df.head())
```

Visualize Data Using Matplotlib:

[python]

```
import matplotlib.pyplot as plt
df.plot(kind='bar', x='product_id', y='total_sales', legend=False)
plt.title('Total Sales by Product')
plt.xlabel('Product ID')
plt.ylabel('Total Sales')
plt.show()
```

Create a Line Plot with Matplotlib:

[python]

```
df.plot(kind='line', x='product_id', y='total_sales')
plt.title('Sales Trend by Product')
plt.show()
```

Generate a Histogram Using Seaborn:

[python]

```
import seaborn as sns
sns.histplot(df['total_sales'], kde=True)
plt.title('Distribution of Total Sales')
plt.show()
```

Filter Data and Analyze Using Pandas:

[python]

```
filtered_df = df[df['total_sales'] > 10000]
print(filtered_df)
```

Analyze Aggregated Data Using Pandas:

[python]

```
grouped = df.groupby('product_id').agg({'total_sales': 'sum'}).reset_index()
print(grouped)
```

Create a Scatter Plot:

[python]

```
sns.scatterplot(x='product_id', y='total_sales', data=df)
plt.title('Sales by Product')
plt.show()
```

Close the Connection:

[python]

```
conn.close()
```

Time-Series and Geospatial Data

Working with time-series data in PostgreSQL

Time-series data refers to data points indexed in time order. PostgreSQL is highly capable of handling time-series data due to its robust support for date and time functions, window functions, and indexing. You can efficiently store, query, and aggregate time-series data, and even work with extensions like TimescaleDB to enhance PostgreSQL's time-series capabilities.

Key Concepts:

- ✓ **Timestamps:** PostgreSQL supports `TIMESTAMP` and `TIMESTAMPTZ` data types for storing dates and times.
- ✓ **Intervals:** PostgreSQL allows you to work with time intervals to calculate differences or add/subtract time periods.
- ✓ **Window Functions:** Useful for time-based calculations like moving averages or cumulative sums.
- ✓ **Indexing:** Time-series data often uses indexes for faster query performance, especially on timestamp columns.

Examples of Working with Time-Series Data:

Create a Table with a Timestamp:

```
CREATE TABLE sensor_data (
    sensor_id INT,
    reading NUMERIC,
    reading_time TIMESTAMPTZ
);
```

Insert Time-Series Data:

```
INSERT INTO sensor_data (sensor_id, reading, reading_time)
VALUES (1, 23.5, '2023-09-14 10:00:00+00'),
       (1, 24.0, '2023-09-14 11:00:00+00'),
       (1, 25.1, '2023-09-14 12:00:00+00');
```

Query Time-Series Data for a Specific Time Range:

```
SELECT * FROM sensor_data
WHERE reading_time BETWEEN '2023-09-14 10:00:00' AND '2023-09-14 12:00:00';
```

Calculate the Difference Between Successive Readings:

```
SELECT sensor_id, reading, reading_time,
       reading - LAG(reading) OVER (ORDER BY reading_time) AS diff
FROM sensor_data
ORDER BY reading_time;
```

Compute a Moving Average:

```
SELECT sensor_id, reading, reading_time,
       AVG(reading) OVER (ORDER BY reading_time ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving_avg
FROM sensor_data;
```

Group Data by Time Intervals:

```
SELECT date_trunc('hour', reading_time) AS hour, AVG(reading) AS avg_reading
FROM sensor_data
GROUP BY hour;
```

Query Data for the Last Week:

```
SELECT * FROM sensor_data
WHERE reading_time >= NOW() - INTERVAL '1 week';
```

Find the First and Last Reading for Each Day:

```
SELECT sensor_id,
       MIN(reading_time) AS first_reading_time,
       MAX(reading_time) AS last_reading_time
FROM sensor_data
GROUP BY sensor_id, date_trunc('day', reading_time);
```

Compute Cumulative Sum Over Time:

```
SELECT sensor_id, reading, reading_time,
       SUM(reading) OVER (ORDER BY reading_time) AS cumulative_sum
FROM sensor_data;
```

Create an Index on the Timestamp Column for Faster Queries:

```
CREATE INDEX idx_sensor_data_time ON sensor_data (reading_time);
```


Introduction to PostGIS for geospatial data

PostGIS is a PostgreSQL extension that adds support for geospatial data. It allows you to store, query, and manipulate geographic objects such as points, lines, and polygons. PostGIS provides spatial data types like POINT, LINESTRING, POLYGON, and a rich set of spatial functions for analysis.

Key Concepts:

- ✓ Geometry Types: PostGIS provides types like POINT, LINESTRING, POLYGON, etc., for storing geometric data.
- ✓ Spatial Queries: Functions like ST_Within(), ST_Intersects(), and ST_Distance() are used for spatial querying.
- ✓ SRID (Spatial Reference System Identifier): Defines the coordinate system for the geometries.
- ✓ Indexes: PostGIS uses GiST indexes for efficient spatial queries.

Examples of Working with PostGIS:

Install PostGIS Extension:

```
CREATE EXTENSION postgis;
```

Create a Table with Geospatial Data:

```
CREATE TABLE locations (  
  id SERIAL PRIMARY KEY,  
  name VARCHAR(100),  
  geom GEOMETRY(POINT, 4326) -- SRID 4326 for WGS 84  
);
```

Insert Geospatial Data (Points):

```
INSERT INTO locations (name, geom)  
VALUES ('Location 1', ST_SetSRID(ST_MakePoint(-73.9857, 40.7484), 4326)), -- Example: New York  
('Location 2', ST_SetSRID(ST_MakePoint(-118.2437, 34.0522), 4326)); -- Example: Los Angeles
```

Query Points Within a Specific Bounding Box:

```
SELECT name, geom  
FROM locations  
WHERE geom && ST_MakeEnvelope(-125, 30, -110, 50, 4326);
```

Calculate the Distance Between Two Points:

```
SELECT ST_Distance(  
  ST_SetSRID(ST_MakePoint(-73.9857, 40.7484), 4326),  
  ST_SetSRID(ST_MakePoint(-118.2437, 34.0522), 4326)  
) AS distance_in_meters;
```

Find Locations Within a Certain Distance:

```
SELECT name  
FROM locations  
WHERE ST_DWithin(  
  geom,  
  ST_SetSRID(ST_MakePoint(-73.9857, 40.7484), 4326),  
  50000 -- Within 50 km  
);
```

Create a Polygon and Query Points Inside It:

```
SELECT name
FROM locations
WHERE ST_Within(
    geom,
    ST_GeomFromText('POLYGON((-73 40, -74 40, -74 41, -73 41, -73 40))', 4326)
);
```

Convert Geometry to GeoJSON Format:

```
SELECT name, ST_AsGeoJSON(geom)
FROM locations;
```

Find Points that Intersect with a Line:

```
SELECT name
FROM locations
WHERE ST_Intersects(
    geom,
    ST_GeomFromText('LINESTRING(-73 40, -118 34)', 4326)
);
```

Create a GiST Index for Geospatial Queries:

```
CREATE INDEX idx_locations_geom ON locations USING GIST (geom);
```

Advanced queries with time-series and geospatial data

Combining time-series and geospatial data allows you to analyze time and space dimensions together. PostGIS and PostgreSQL provide powerful ways to handle such queries efficiently, allowing for spatio-temporal analytics.

Examples of Advanced Queries with Time-Series and Geospatial Data:

Find Sensor Readings Within a Time Range and Location:

```
SELECT sensor_id, reading, reading_time
FROM sensor_data sd
JOIN locations l ON ST_DWithin(sd.geom, l.geom, 10000) -- 10 km radius
WHERE reading_time BETWEEN '2023-09-01' AND '2023-09-14';
```

Calculate the Distance Traveled by a Moving Object Over Time:

```
SELECT sensor_id,
    SUM(ST_Distance(geom, LAG(geom) OVER (ORDER BY reading_time))) AS total_distance
FROM sensor_data
WHERE sensor_id = 1
GROUP BY sensor_id;
```

Find the Nearest Location for Each Sensor Reading:

```
SELECT sd.sensor_id, l.name AS nearest_location,
    ST_Distance(sd.geom, l.geom) AS distance
FROM sensor_data sd
JOIN locations l ON ST_Distance(sd.geom, l.geom) < 10000 -- 10 km radius
ORDER BY distance;
```

Group Readings by Hour and Location:

```
SELECT date_trunc('hour', sd.reading_time) AS hour, l.name, AVG(sd.reading) AS avg_reading
FROM sensor_data sd
JOIN locations l ON ST_Within(sd.geom, l.geom)
GROUP BY hour, l.name;
```

Create a Heatmap of Readings Over a Geographic Area:

```
SELECT ST_AsHeatmap(geom)
FROM sensor_data
WHERE reading_time >= '2023-09-01' AND reading_time <= '2023-09-14';
```

Find All Readings Within a Polygon Over a Certain Time Period:

```
SELECT sd.sensor_id, sd.reading, sd.reading_time
FROM sensor_data sd
WHERE ST_Within(
    sd.geom,
    ST_GeomFromText('POLYGON((-73 40, -74 40, -74 41, -73 41, -73 40))', 4326)
) AND reading_time BETWEEN '2023-09-01' AND '2023-09-10';
```

Calculate the Speed of an Object Based on Time and Distance:

```
SELECT sensor_id,
    ST_Distance(geom, LAG(geom) OVER (ORDER BY reading_time)) /
    EXTRACT(EPOCH FROM reading_time - LAG(reading_time) OVER (ORDER BY reading_time)) AS speed
FROM sensor_data;
```

Identify Hotspots for Sensor Readings Over Time:

```
SELECT ST_ClusterKMeans(geom, 5) AS cluster_id, COUNT(*)
FROM sensor_data
WHERE reading_time >= '2023-09-01'
GROUP BY cluster_id;
```

Find the Closest Weather Station to a Sensor Based on Location and Time:

```
SELECT sensor_id, ws.station_id,
    ST_Distance(sd.geom, ws.geom) AS distance,
    ws.temperature
FROM sensor_data sd
JOIN weather_stations ws ON ST_DWithin(sd.geom, ws.geom, 10000)
WHERE sd.reading_time = ws.record_time;
```

Track Movement Across Geographic Locations Over Time:

```
SELECT sensor_id, reading_time, geom,
    LEAD(geom) OVER (PARTITION BY sensor_id ORDER BY reading_time) AS next_location
FROM sensor_data;
```