PostgreSQL

ADV. DEVELOPER COURSE

Custom PostgreSQL Online Training conducted at IBM

Mohammed Abdul Sami

sabdul2983@gmail.com

DBA and Consultant

Index

Introduction

This Document is created as reference material for PostgreSQL Developer Course conducted at IBM Inc. Course focuses primarily on understanding Architecture, SQL Implementation in PostgreSQL, PL/pgSQL programming on PostgreSQL and Performance Tuning.

About Trainer

Mohammed Abdul Sami is an experienced PostgreSQL DBA & System Administrator having more than 24+ years of experience in managing Database Servers and Windows / Unix / Linux Servers. Skilled at scripting and automation.

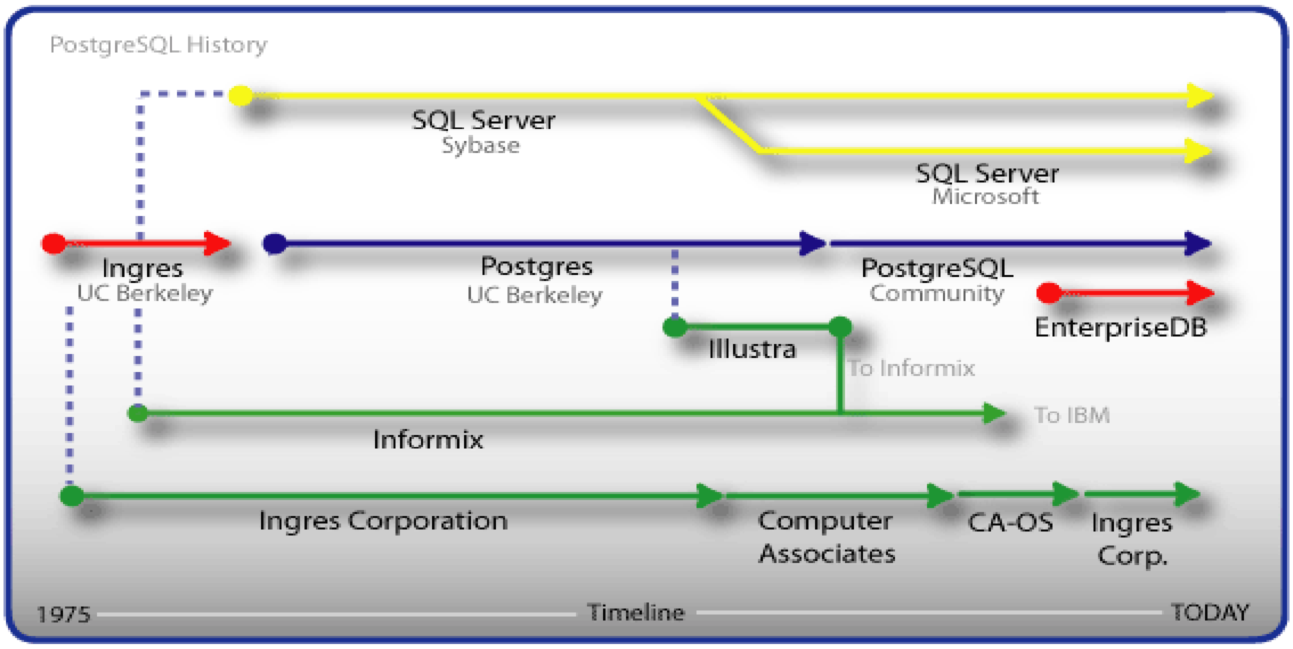
Since year 2015 he is working as freelance consultant and trainer and is a frequent faculty at Oracle University for MySQL, Solaris etc courses.

Lesson 1 : Introduction to PostgreSQL

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2, developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later. PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:

- complex queries
- foreign keys
- triggers
- updatable views
- transactional integrity
- multiversion concurrency control (MVCC)

Brief History of PostgreSQL



The object-relational database management system now known as PostgreSQL is derived from the POST-GRES package written at the University of California at Berkeley. With over two decades of development behind it, PostgreSQL is now the most advanced open-source database available anywhere.

Database Limits

Item	Upper Limit	Comment
database size	unlimited	

number of databases	4294950911	
relations per database	1431650303	
relation size	32 TB	with the default BLCKSZ of 8192 bytes
rows per table	limited by the number of tuples that can fit onto 4,294,967,295 pages	
columns per table	1600	further limited by tuple size fitting on a single page.
columns in a result set	1664	
field size	1 GB	
identifier length	63 bytes	can be increased by recompiling PostgreSQL
indexes per table	unlimited	constrained by maximum relations per database
columns per index	32	can be increased by recompiling PostgreSQL
partition keys	32	can be increased by recompiling PostgreSQL

Common Database Object Names

Table or Index	Relation
Row	Tuple
Column	Attribute

Data Block	Page (when block is on disk)
Page	Buffer (when block is in memory)

PostgreSQL Database cluster

A single PostgreSQL instance is referred as a PostgreSQL Database Cluster as it can support multiple databases.

Major Features of PostgreSQL

- Portable:
- Written in ANSI C
- Supports Windows, Linux, Mac OS/X and major UNIX platforms
- Reliable:
- ACID Compliant
- Supports Transactions
- Supports Save points
- Uses Write Ahead Logging (WAL)
- Scalable:
- Uses Multi-version Concurrency Control
- Supports Table Partitioning
- Supports Tablespaces
- Secure:
- Employs Host-Based Access Control
- Provides Object-Level Permissions
- Supports Logging
- SSL
- Recovery and Availability:
- Streaming Replication
- Replication Slots
- Supports Hot-Backup, pg_basebackup
- Point-in-Time Recovery
- Advanced:
- Supports Triggers and Functions
- Supports Custom Procedural Languages PL/pgSQL, PL/Perl, PL/TCL, PL/PHP, PL/Java
- Upgrade using pg_upgrade

- Unlogged Tables
- Materialized Views

PostgreSQL Editions

PostgreSQL Community Edition : Released under PostgreSQL license which is a liberal Open Source license, similar to the BSD or MIT licenses

EDB PostgreSQL : Enterprise ready commercial PostgreSQL.

PostgreSQL in Public Cloud

All most all could service providers are offering fully managed PostgreSQL as DBAAS service. Listed some of the provers

EDB BigAnimal

It is a fully managed PostgreSQL database as service (DBaaS) provided by EnterpriseDB.

RDS for Postgres

RDS for Postgres is provided as database as service from Amazon public cloud. It is self managed database easy & quick to deploy and minimal management is required

Aurora Database for Postgres

Aurora for postgres is another database as service from amazon cloud which allows you to deploy fully managed multi-master cluster.

Azure Database for PostgreSQL

It is a fully managed PostgreSQL database service provided by Microsoft Azure platform.

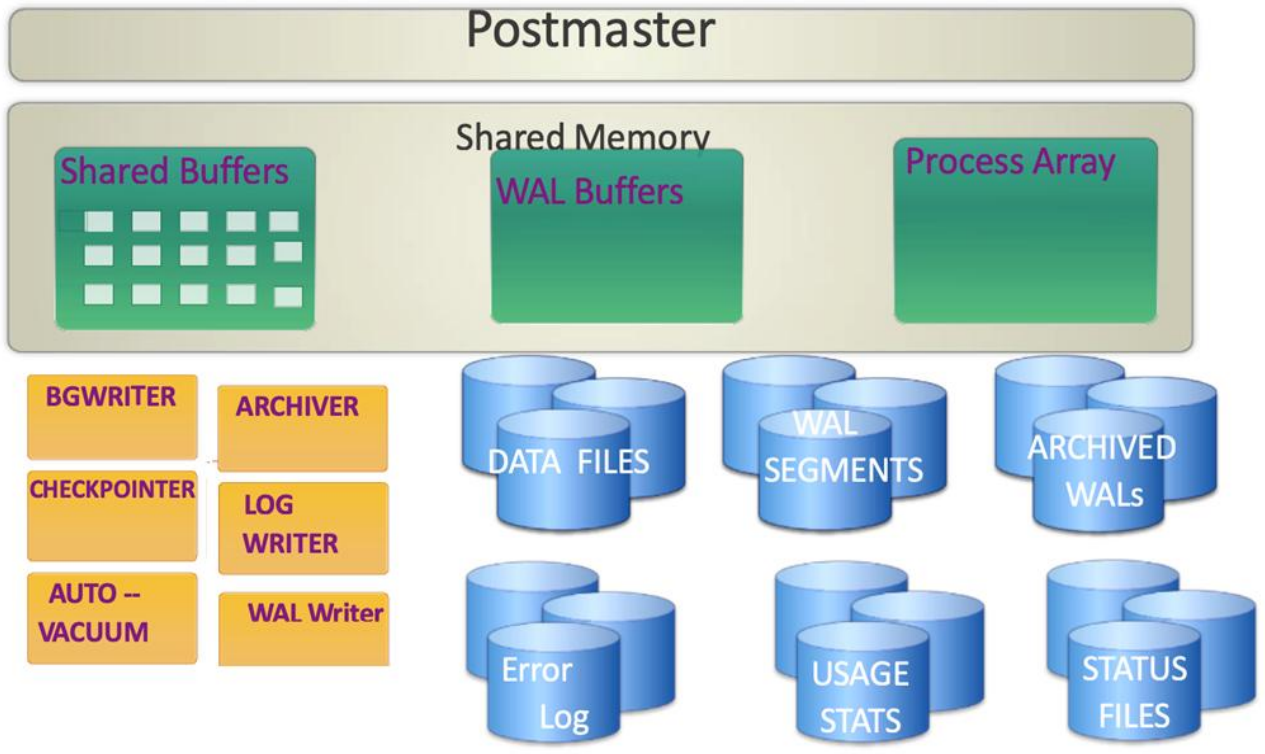
Cloud SQL for PostgreSQL

Is a fully managed RDBMS postgresql compatible database from Google Cloud.

Lesson 2 - Architecture

- PostgreSQL uses processes, not threads
- Postmaster process acts as supervisor
- Several utility processes perform background work
- postmaster starts them, restarts them if they die
- One backend process per user session
- Postmaster listens for new connections

Process and Memory Architecture



Postmaster

- Postmaster is the master process called Postgres
- Manages Utility processes, restarts them if they die.
- Listens on 1-and-only-1 tcp port (default : 5432)
- Receives client connection request

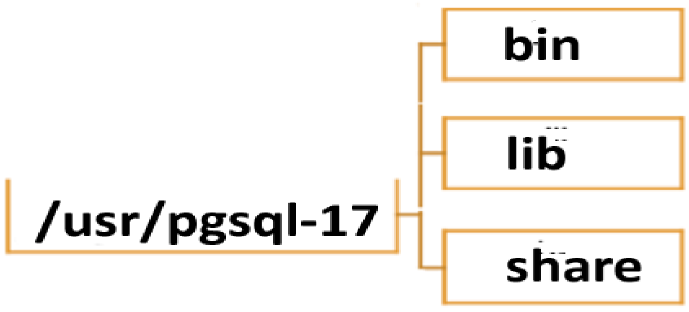
Utility Processes

- **Background writer** → Writes updated data blocks to disk
- **WAL writer** → Flushes write-ahead log to disk

- **Checkpointer process** → Automatically performs a checkpoint based on config parameters
- **Logging collector** → Routes log messages to syslog, eventlog, or log files
- **Autovacuum launcher** → Starts Autovacuum workers as needed
- **Autovacuum workers** → Recover free space for reuse
- **Archiver** → Archives write-ahead log files

Storage Layout on RHEL

Installation Directory Layout



- **bin** → Binary files (all postgres commands and executables)
- **lib** → Library files (.so files)
- **share** → Extensions and sample config files

NOTE: All the Installation Files are owned by Linux "root"user

Data Directory Layout

Data Directory is located in the home directory of Linux service user "postgres" and usually it is /var/lib/pgsql/17/data

Item	Description
PG_VERSION	A file containing the major version number of PostgreSQL
base	Subdirectory containing per-database subdirectories (pg_default tablespace)
current_logfiles	File recording the log file(s) currently written to by the logging collector
global	Subdirectory containing cluster-wide tables, such as pg_database

pg_commit_ts	Subdirectory containing transaction commit timestamp data
pg_dynshmem	Subdirectory containing files used by the dynamic shared memory subsystem
pg_logical	Subdirectory containing status data for logical decoding
pg_multixact	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify	Subdirectory containing LISTEN/NOTIFY status data
pg_replslot	Subdirectory containing replication slot data
pg_serial	Subdirectory containing information about committed serializable transactions
pg_snapshots	Subdirectory containing exported snapshots
pg_stat	Subdirectory containing permanent files for the statistics subsystem
pg_stat_tmp	Subdirectory containing temporary files for the statistics subsystem
pg_subtrans	Subdirectory containing subtransaction status data
pg_tblspc	Subdirectory containing symbolic links to tablespaces
pg_twophase	Subdirectory containing state files for prepared transactions
pg_wal	Subdirectory containing WAL (Write Ahead Log) files
pg_xact	Subdirectory containing transaction commit status data
pg_hba.conf	File used to store Server access ACLs
pg_ident.conf	User name maps are defined in this file and used by Authentication methods such as IDENT, GSSAPI
postgresql.conf	A file used for storing configuration parameters
postgresql.auto.conf	A file used for storing configuration parameters that are set by ALTER SYSTEM

postmaster.opts	A file recording the command-line options the server was last started with
postmaster.pid	A lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (empty on Windows), first valid listen_address (IP address or *, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown)

Note: All directories and files in “data” directory are owned by linux “postgres” user and under same user posgreSQL service run

Write Ahead Logging (redo) and Undo (Rollback)

Write a head logging

- PostgreSQL uses Write-Ahead Logging (WAL) as its approach to transaction logging.
- WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been written to WAL files.
- No need to flush data pages to disk on every transaction commit.
- In the event of a crash we will be able to recover the database using the logs
- This is roll-forward recovery, also known as REDO

Rollback and Undo

- Dynamic allocation of disk space is used for the storage and processing of records within tables.
- The files that represent the table grow as the table grows.
- It also grows with transactions that are performed against it.
- The data is stored within the file that represents the table.
- When deletes and updates are performed on a table, the file that represents the object will contain the previous data
- This space gets reused, but need to force recovery of used space. (Using Vaccum)

Data File Storage Internals

- File-per-table, file-per-index.
- A table-space is a directory.
- Each database that uses that table-space gets a subdirectory.
- Each relation using that tablespace/database combination gets one or more files, in 1GB chunks.
- Additional files used to hold auxiliary information (free space map, visibility map).

- Each file name is a number (see pg_class.relfilenode)

TOAST

PostgreSQL does not allow rows to span multiple pages (disk blocks) and it commonly uses 8k page size. This makes it impossible to store oversized columns. To over come this PostgreSQL introduced TOAST (The Oversize Attribute Storage Technique). TOAST uses a separate data file on disk to save oversized columns.

TOAST allows transparent handling of columns which does not fit into the Page (block) by saving them in TOAST file on disk. Handling TOAST storage and retrieval does not require any special techniques, normal queries will take care of it.

Manually Specifying TOAST storage

We can force a column to be saved in TOAST file rather the Table data file by specifying the STORAGE property for the column while creating table or later by altering table.

Syntax:

```
CREATE TABLE <table name> (name varchar(40), photo bytea STORAGE  EXTERNAL,…)
CREATE TABLE <table name> (name varchar(40), photo bytea STORAGE  EXTENDED COMPRESSED,…)
```

Note: Fixed size columns like int, numeric etc cannot be saved in TOAST.

Free Space and Visibility Map Files

- Each Relation has a free space map
    - Stores information about free space available in the relation
    - File named as filenode number plus the suffix _fsm
- Tables also have a visibility map
    - Track which pages are known to have no dead tuples
    - Stored in a fork with the suffix _vm

Lesson 3 - PostgreSQL Clients

psql Client (default client)

psql is default command line client program for postgresql database on both Unix/Linux and Windows OSes it can be invoked locally or from a remote client machine.

```
# psql  -U <username>  -d  <database name>
```

If server demands the password psql prompts for password other wise it connects to the specified database

Other options of PSQL are below

-c command

--command=command

Specifies that psql is to execute one command string, command, and then exit. This is useful in shell scripts.

-f filename

--file=filename

Use the file filename as the source of commands instead of reading commands interactively. After the file is processed, psql terminates.

-h hostname

--host=hostname

Specifies the host name of the machine on which the server is running.

-p port

--port=port

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the PGPORT environment variable or, if not set, to the port specified at compile time, usually 5432.

-1

--single-transaction

When psql executes a script with the -f option, adding this option wraps BEGIN/COMMIT around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

\x   Expanded display is on/off

Extracting Information from Command Line

```
\d [ table ]
```

List tables in the database, or if table is specified, list the columns in that table. If table name is specified as an asterisk (``*''), list all tables and column information for each tables.

`\da` List all available aggregates.

`\dd object`

List the description from pg_description of the specified object, which can be a table, table.column, type, operator, or aggregate.

`\df` List functions.

`\di` List only indexes.

`\do` List only operators.

`\ds` List only sequences.

`\dS` List system tables and indexes.

`\dt` List only non-system tables.

`\dT` List types.

`\l` List all the databases in the server.

`\x` Toggles extended row format mode.

`\w filename`

Outputs the current query buffer to the file file name.

`\z` Produces a list of all tables in the database with their appropriate ACLs (grant/revoke permissions) listed.

`\! [ command ]`

Escape to a separate Unix shell or execute the Unix (OS) command command.

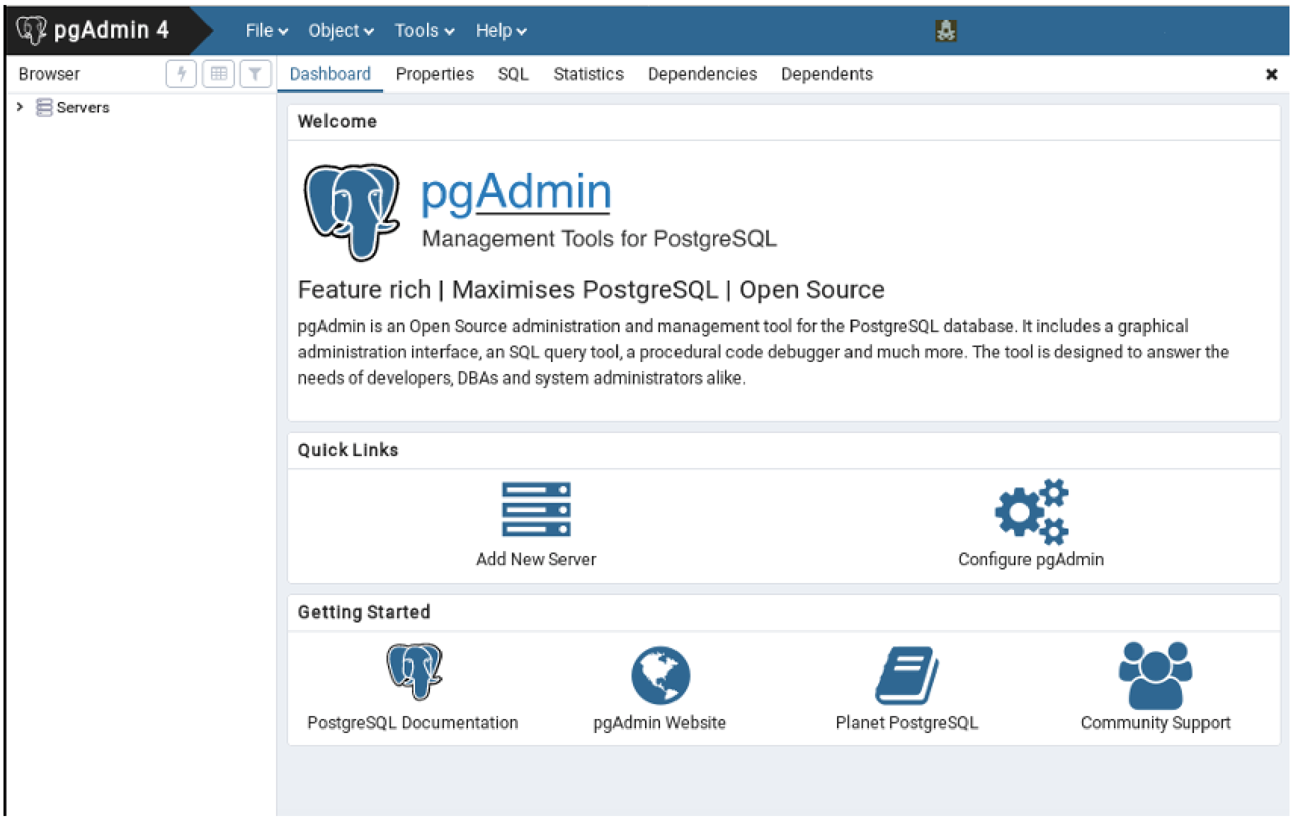
`\?` Get help information about the slash (``\'') commands.

Setting Variables in psql

```
\set   <var>   <values>
eg:
postgres=# \set tbn emp
postgres=# select * from :tbn;
postgres=# \unset tbn
```

GUI Client PgAdmin4

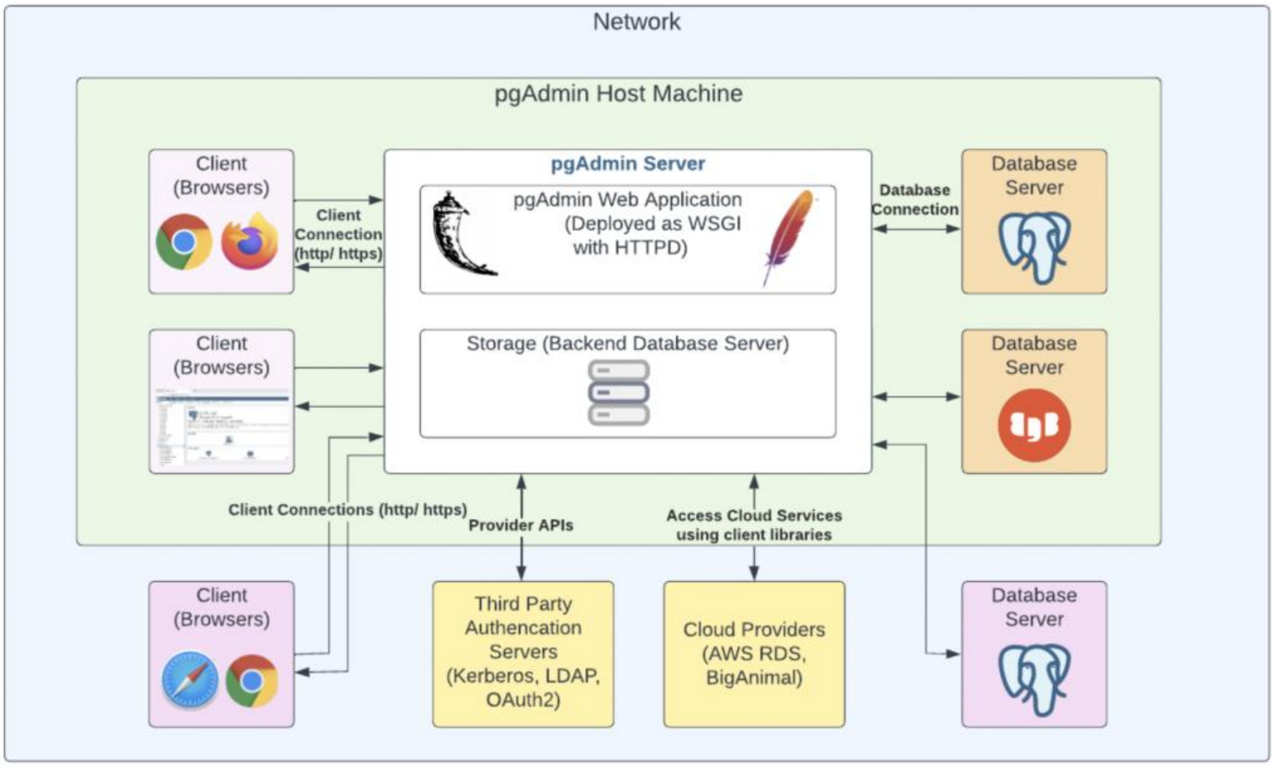
PostgreSQL provides PgAdmin as default graphical client for managing PostgreSQL database. Almost all administration task can be performed from PgAdmin interface except for restarting the reserver or editing postgresql.conf / pg_hba.conf files



pgAdmin is an Open Source administration and management tool for the PostgreSQL database. It includes a graphical administration interface, an SQL query tool, a procedural code debugger and much more. The tool is designed to answer the needs of developers, DBAs and system administrators alike.

- Available on MacOS, Windows and Linux
- Supports Desktop mode
- Supports in Web-Server mode and act as central GUI client fo many PostgreSQL Instances / RDS for PostgreSQL instance.
- GUI Interface is same for both Desktop and Web-Server modes

PGAdmin4 Web-Server Architecture


Network
pgAdmin Host Machine
Client (Browsers)
Client Connection (http/ https)
pgAdmin Server
pgAdmin Web Application (Deployed as WSGI with HTTPD)
Storage (Backend Database Server)
Database Connection
Database Server
Client (Browsers)
Database Server
Client Connections (http/ https)
Provider APIs
Access Cloud Services using client libraries
Client (Browsers)
Third Party Authencation Servers (Kerberos, LDAP, OAuth2)
Cloud Providers (AWS RDS, BigAnimal)
Database Server

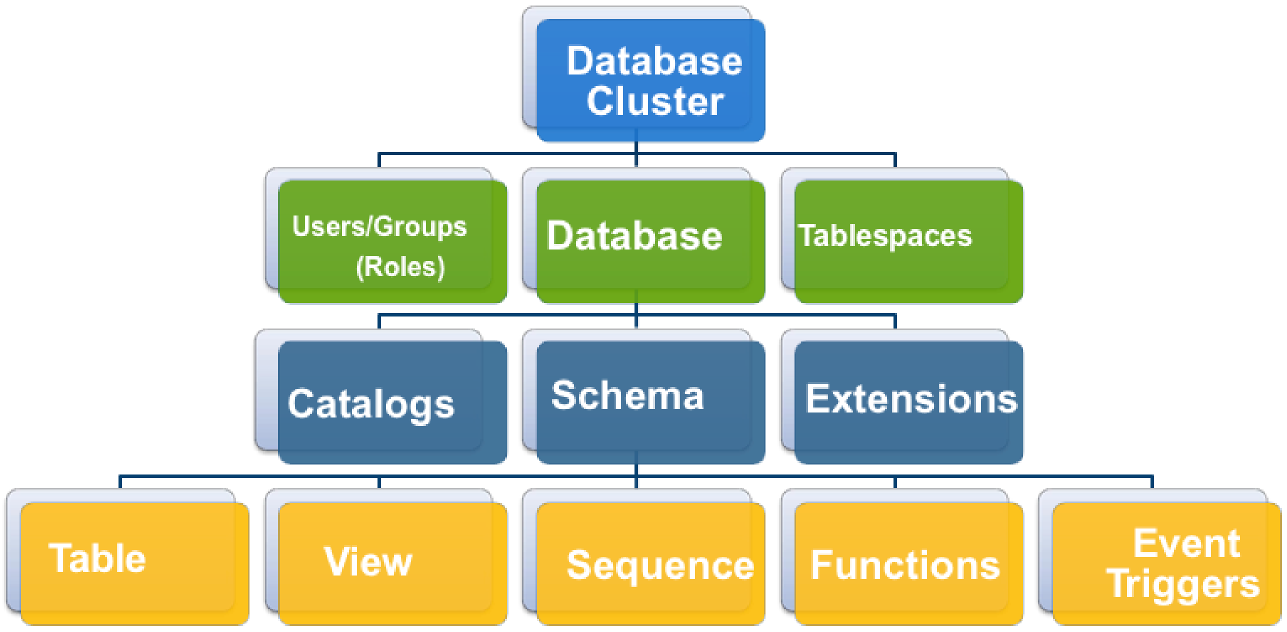
A central Web-Server based PGAdmin4 running on a linux machine and the DBA/Developers connect through a Browser.

- Uses SQLLite database for storing metadata
- Requires authentication to access PGAdmin4
- Ability to store multiple PostgreSQL db connection details for each PGAdmin4 user
- Centralized access to PostgreSQL databases for DBAs and Developers

Lesson 4 - Creating and Managing Databases

Object Hierarchy

Below Diagram show the hierarchy of object in a PostgreSQL Database Cluster.



Database

- A running PostgreSQL server can manage multiple databases.
- A database is a named collection of SQL objects.
- It is a collection of schemas and the schemas contain the tables, functions, etc.
- Databases are created with CREATE DATABASE command.
- Databases are destroyed with DROP DATABASE command.

List the existing databases:

```
SQL: SELECT datname FROM pg_database;
PSQL META COMMAND: \l (backslash lowercase L)
```

Creating a Database

There is a program that you can execute from the shell to create new databases, createdb.

```
$ createdb <dbname>
```

Create Database SQL command can be used to create a database in a cluster.

Syntax:

```
postgres=#  CREATE DATABASE name
          [ [ WITH ] [ OWNER [=] user_name ]
          [ TEMPLATE [=] template ]
          [ ENCODING [=] encoding ]
          [ LC_COLLATE [=] lc_collate ]
```

```
           [ LC_CTYPE [=] lc_ctype ]
           [ TABLESPACE [=] tablespace_name ]
           [ ALLOW_CONNECTIONS [=] allowconn ]
           [ CONNECTION LIMIT [=] connlimit ]
           [ IS_TEMPLATE [=] istemplate ] ]
```

by default every created database is accessible to all user as it is part of "public" role as a precaution we have to remove the CONNECT privilege from public to prevent any unauthorized access to the database.

```
postgres=#  REVOKE  CONNECT  on <database>  FROM PUBLIC;
```

Now only owner or Superuser can connect to this database.

Switching to a Database:

```
postgres=#  \connect   <dbname>
```

(meta shortcut \c can also be used instead of \connect).

Connecting to a Database:

We can connect to database using psql client

```
$  psql  -U <USer>   <dbname>
dname=#
```

Querying How Many Database Exists:

We can query pg_database system table to find the list of databases in the cluster.

```
postgres=#  select datname from pg_database;
```

Or we can also use psql meta command to find the list of databases created so far.

```
postgres=#  \l
```

Dropping Database:

```
postgres=# DROP DATABASE  <dbname>;
```

(only Owner or postgres superuser can drop a database)

Schema

- A database can contains one or more named schemas.
- By Default, all database contain public schema.
- There are several reasons why one might want to use schemas:
- To allow many users to use one database without interfering with each other.
- To organize database objects into logical groups to make them more manageable.
- Third-party applications can be put into separate schemas so they cannot collide with the names of other objects.

Relation Between USER / SCHEMA and OBJECTS

User Owns Schema and Schema contains the database objects like table, views etc.

Creating Schema

A Schema can be created by connecting to the desired database and running the CREATE SCHEMA command.

Syntax:

```
     CREATE SCHEMA  <schema name>  AUTHORIZATION  <username>;
```

For example if we need to create a schema “raj” in PROD database for user ‘raj’ then run the following commands

```
postgres=#  \c  prod
prod=#  CREATE SCHEMA  raj AUTHORIZATION  raj;
```

Now a new schema called “raj” is created in the database PROD which is owned by user “raj”

Schema Search Path

Object in the database should be referred as <schema_name>.<object_name> however it is tedious to give qualified names. To make this easier we can set SCHEMA SEARCH PATH so that the objects in the schemas mentioned in SCHEMA SEARCH PATH can be accessed without qualified names.

To Check the Current Search PATH

To see the current search path use SHOW Command

```
postgres=# SHOW search_path;
```

To SET the search path use SET command

```
postgres=# SET search_path TO myschema, public, raj;
postgres=# ALTER USER <username> IN DATABASE <db name> SET = myschema, public;
```

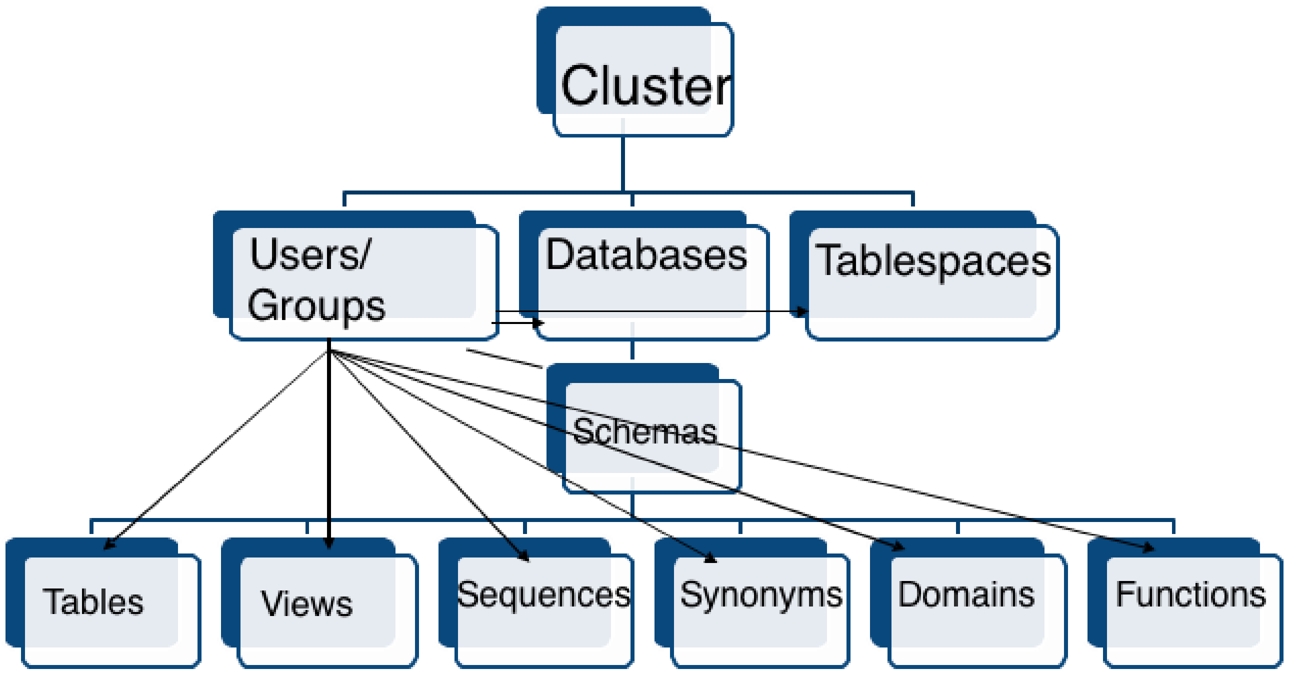
Listing Schemas

```
postgres=# \dn
postgres=# select * from pg_namespace;
```

Database Objects

- Database Schemas can contain different types of objects including:
- Tables
- Sequences
- Views
- Synonyms (only in Commercial PostgreSQL)
- Domains
- Packages
- Functions
- Procedures

Object Ownership



Data dictionary Objects

`pg_database` -- All database details are stored in this table

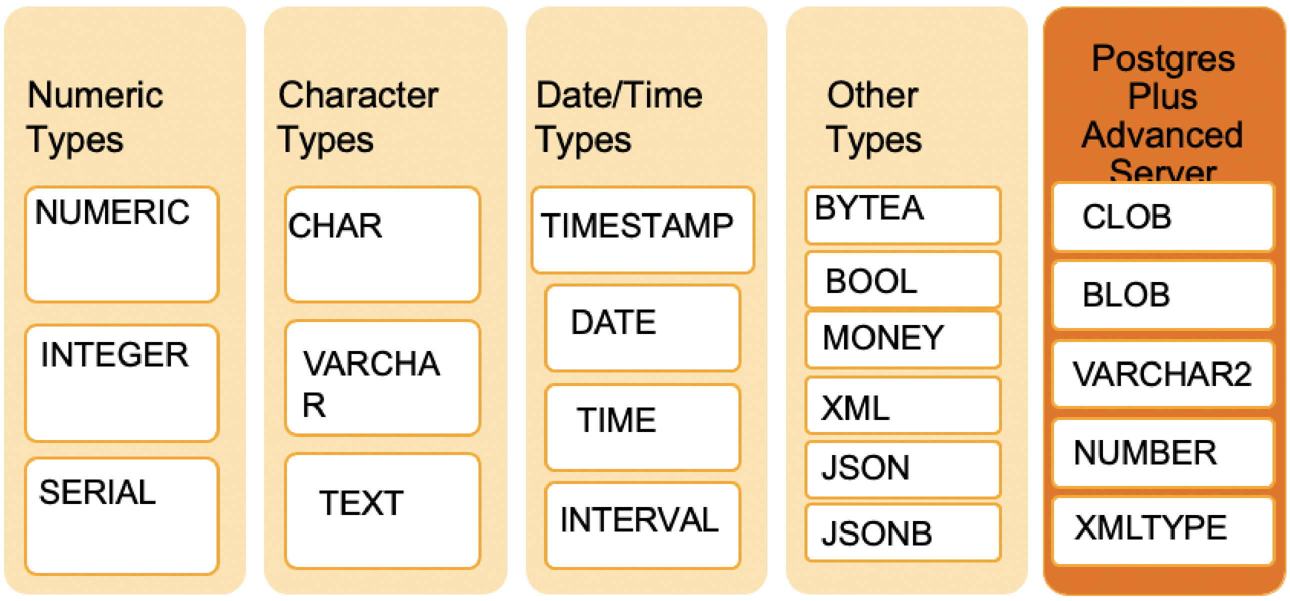
`pg_namespace` – details of all schemas.

`pg_class` – Details of all tables and table like objects.

`pg_tables` -- Details of all tables

Lesson 5 - DDL & DML Statements

Normal Data Types



Numeric Data Type

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
Integer, int	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
smallserial	2 bytes	small autoincrementing integer	1 to 32767
serial	4 bytes	autoincrementing integer	1 to 2147483647

Name	Storage Size	Description	Range
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Monetary Types

The money type stores a currency amount with a fixed fractional precision. The fractional precision is determined by the database's `lc_monetary` setting.

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

Character Types

Name	Description
character varying(***n***), varchar(***n***)	variable-length with limit
character(***n***), char(***n***), bpchar(***n***)	fixed-length, blank-padded
bpchar	variable unlimited length, blank-trimmed
text	variable unlimited length

Date/Time Types

Name	Storage Size	Description	Low Value	High Value
timestamp [(*p*)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD
timestamp [(*p*)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD
date	4 bytes	date (no time of day)	4713 BC	5874897 AD
time [(*p*)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00
time [(*p*)] with time zone	12 bytes	time of day (no date), with time zone	00:00:00+1559	24:00:00-1559

Name	Storage Size	Description	Low Value	High Value
interval [***fields***] [(***p***)]	16 bytes	time interval	-178000000 years	178000000 years

Where (***p***) specifies the number of fractional digits retained in the seconds field and range is 0-6.

Interval field values:

YEAR

MONTH

DAY

HOUR

MINUTE

SECOND

Binary Data Types

Name	Storage Size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

Boolean Data Type

Name	Storage Size	Description	Valid Values
boolean	1 byte	state of true or false	True, yes,1,on False,no,0,off

Enumerated Types

We can create a new data type associated with a specific list of values. When any column declared with this type then it will accept values from given list.

Create ENUM Type:

Syntax:

```
CREATE TYPE <typename>  AS ENUM ('v1', 'v2', 'v3',….'vn');
```

Example:

```
CREATE TYPE weekdays AS ENUM ('Monday', 'Tuesday', 'Wednesday','Thursday', 'Friday');
```

DDL Statements

Data Definition Language allows us to define & create objects and when needed make changes to schemas or drop objects.

Creating Tables

- A table is a named collection of rows
- Each table row has same set of columns
- Each column has a data type and optionally a constraint
- Tables can be created using the CREATE TABLE statement

Syntax:

```
CREATE [TEMPORARY][UNLOGGED] TABLE table_name
( [column_name data_type [ column_constraint], ..... )
[ INHERITS ( parent_table) ]
[ TABLESPACE tablespace_name ]
[ USING INDEX TABLESPACE tablespace_name ]
```

Constraints

Constraints are used to enforce data integrity

- PostgreSQL supports different types of constraints:
  - NOT NULL
  - CHECK
  - UNIQUE
  - PRIMARY KEY
  - FOREIGN KEY
  - EXCLUDE
- Constraints can be defined at the column level or table level
- Constraints can be added to an existing table using the ALTER TABLE statement
- Constraints can be declared DEFERRABLE or NOT DEFERRABLE
- Constraints prevent the deletion of a table if there are dependencies
- Exclusion constraint restricts the overlapping entries (btree_gist)

Rules

- Rules are some where between Constraint and Trigger
- Like Triggers, they also fires on DML events
- Can be created for Tables & Views
- Like Triggers, they also sees NEW and OLD data structures

```
Syntax:
CREATE [OR REPLACE] RULE rule_name AS
    ON event
    TO table_name
    [WHERE condition]
    DO [ALSO | INSTEAD] action;
```

rule_name: The name of the rule.

event: One of SELECT, INSERT, UPDATE, or DELETE.

table_name: The table or view on which the rule applies.

condition: Optional condition that must be satisfied for the rule to apply.

action: One or more SQL commands to be executed when the rule is triggered. Use ALSO to execute the original action as well, or INSTEAD to replace it.

Example:

```
CREATE TABLE logs_2024 (
    id serial PRIMARY KEY,
    message text,
    created_at timestamp DEFAULT NOW()
);
CREATE VIEW logs AS SELECT * FROM logs_2024;
CREATE RULE logs_insert AS
    ON INSERT TO logs
    DO INSTEAD
    INSERT INTO logs_2024 (message)
    VALUES (NEW.message);
```

ALTER TABLE

ALTER TABLE is used to modify the structure of a table

Can be used to:

- Add a new column
- Modify an existing column
- Add a constraint
- Drop a column
- Change schema

- Rename a column, constraint or table
- Enable and disable triggers and rules on a table Change ownership
- Enable or disable logging for a table

Syntax:

```
ALTER TABLE table_name ADD COLUMN column_name datatype column_constraint;
ALTER TABLE table_name  DROP COLUMN column_name;
ALTER TABLE table_name  RENAME COLUMN column_name  TO new_column_name;
ALTER TABLE table_name ALTER COLUMN column_name [SET DEFAULT value | DROP DEFAULT];
ALTER TABLE table_name ALTER COLUMN column_name [SET NOT NULL| DROP NOT NULL];
ALTER TABLE table_name ADD CHECK expression;
ALTER TABLE table_name ADD CONSTRAINT constraint_name constraint_definition;
ALTER TABLE table_name RENAME TO new_table_name;
```

DROP TABLE

The DROP TABLE statement is used to drop a table All data and structure is deleted

Related indexes and constraints are dropped

Syntax:

```
DROP TABLE [ IF EXISTS ] name [, ...]
```

DML Statements

Inserting Data

The INSERT statement is used to insert one or multiple rows in a table

Syntax:

```
INSERT INTO table_name [ ( column_name [, ...] ) ]
{ DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
```

Update Data

The UPDATE statement is used to modify zero or more rows in a table

The number of rows modified depends on the WHERE condition

Syntax:

```
UPDATE table_name
SET column_name = { expression | DEFAULT } [ WHERE condition]
```

Deleting Data

The DELETE statement is used to delete one or more rows of a table

Syntax:

```
DELETE FROM [ ONLY ] table_name [ WHERE condition ]
```

Selecting Data

The SELECT statement can be used to retrieve (select) data from a table or an expression. It is one of the most complex statement in SQL language with a lot of clauses to perform data retrieval and transformation.

The SELECT statement has the following clauses:

- Select distinct rows using DISTINCT operator.
- Sort rows using ORDER BY clause.
- Filter rows using WHERE clause.
- Select a subset of rows from a table using LIMIT or FETCH clause.
- Group rows into groups using GROUP BY clause.
- Filter groups using HAVING clause.

Syntax:

```
SELECT [DISTINCT] [columns selection] [FROM <table>] [WHERE <expression>] [GROUP BY <Expression> [HAVING <Expression>]] [ORDER BY <sort expression> ASC | DESC] [LIMIT row_count OFFSET row_to_skip]
```

Aliasing

Alias is a temporary name given to a table or a column during execution of the query.

Below are the various ways of using an alias.

Column Alias

- Column aliases can be used in the SELECT list of a SQL query in PostgreSQL.
- Like all objects, aliases will be in lowercase by default. If mixed-case letters or special symbols, or spaces are required, quotes must be used.
- Column aliases can be used for derived columns.
- Column aliases can be used with GROUP BY and ORDER BY clauses.
- We cannot use a column alias with WHERE and HAVING clauses.

Table Alias

- Table aliases can be used in SELECT lists and in the FROM clause to show the complete record or selective columns from a table.
- Table aliases can be used in WHERE, GROUP BY, HAVING, and ORDER BY clauses.
- When we need data from multiple tables, we need to join those tables by qualifying the columns using table name/table alias.
- The aliases are mandatory for inline queries (queries embedded within another statement) as a data source to qualify the columns in a select list.

Example:

```
SELECT ename employee_name, empno AS employee_id FROM emp;
```

Quoting

- Single quotes and dollar quotes are used to specify non-numeric values

Example:

```
'hello world'
'2011-07-04 13:36:24' '{1,4,5}'
$$A string "with" various 'quotes' in.$$
$foo$A string with $$ quotes in $foo$
```

- Double quotes are used for names of database objects which either clash with keywords, contain mixed case letters, or contain characters other than a-z, 0-9 or underscore

Example:

```
select * from "select"
create table "HelloWorld" ...;
select * from "Sales Data";
```

Built - In Functions & Operators

PostgreSQL provides lot of built-in functions for various needs

Numeric Functions

Function	Description	Example	Result
ABS	Calculate the absolute value of a number	ABS(-10)	10
CBRT	Calculate the cube root of a number	CBRT(8)	2
CEIL	Round a number up to the nearest integer, which is greater than or equal to number	CEIL(-12. 8)	-12
DIV	Return the integer quotient of two numeric values	DIV(8,3)	2
EXP	Return the exponential value in scientific notation of a number	EXP(1)	2.7182818
FLOOR	Round a number down to the nearest integer, which is less than or equal to number	FLOOR(1 0.6)	10
LN	Return the natural logarithm of a numeric value	LN(3)	1.0986122

LOG	Return the base 10 logarithm of a numeric value	LOG(100 0)	3
LOG	Return the logarithm of a numeric value to a specified base	LOG(2, 64)	6
MOD	Divide the first parameter by the second one and return the remainder	MOD(10,4)	1
PI	Return the value of PI	PI()	3.1415926
POWER	Raise a numeric value to the power of a second numeric value	POWER(5, 3)	125
RADIANS	Convert degrees to radians	RADIANS (60)	1.0471975
ROUND	Round a number to the nearest integer or to a specified decimal places	ROUND(1 0.3)	10
SCALE	Return the number of decimal digits in the fractional part	SCALE(1. 234)	3
SIGN	Return the sign (positive, negative) of a numeric value	SIGN(-1)	-1
SQRT	Return the square root of a numeric value	SQRT(3.0)	1.7320508
TRUNC	Truncate a numeric value to a whole number of to the specified decimal places	TRUNC(1 2.3)	12
RANDOM	Return a random number that ranges from 0 to 1	random()	1 0.9684356

String Functions

Function	**Description**	**Example**	**Result**
ASCII	Return the ASCII code value of a character or Unicode code point of a UTF8 character	ASCII(‘A’)	65
CHR	Convert an ASCII code to a character or a Unicode code point to a UTF8 character	CHR(65)	A’
CONCAT	Concatenate two or more strings into one	CONCAT(‘A’,’B’,’C’)	ABC’
CONCAT_WS	Concatenate strings with a separator	CONCAT_WS(‘,’,’A’,’B’,’ C’)	A,B,C’

INITCAP	Convert words in a string to title case	INITCAP('hI tHERE')	Hi There
LEFT	Return the first n character in a string	LEFT('ABC',1)	A'
LENGTH	Return the number of characters in a string	LENGTH('ABC')	3
LOWER	Convert a string to lowercase	LOWER('hI tHERE')	hi there'
LPAD	Pad on the left a a string with a character to a certain length	LPAD('123', 5, '00')	00123'
LTRIM	Remove the longest string that contains specified characters from the left of the input string	LTRIM('00123')	123'
MD5	Return MD5 hash of a string in hexadecimal	MD5('ABC')	
POSITION	Return the location of a substring in a string	POSTION('B' in 'A B C')	3
REPEAT	Repeat string the specified number of times	REPEAT('*', 5)	*****'
REPLACE	Replace all occurrences in a string of substring from with substring to	REPLACE('ABC','B','A')	AAC'
REVERSE	Return reversed string.	REVERSE('ABC')	CBA'
RIGHT	Return last n characters in the string. When n is negative, return all but first |n| characters.	RIGHT('ABC', 2)	BC'
RPAD	Pad on the right of a string with a character to a certain length	RPAD('ABC', 6, 'xo')	ABCxox'
RTRIM	Remove the longest string that contains specified characters from the right of the input string	RTRIM('abcxxzx', 'xyz')	abc'
SUBSTRING	Extract a substring from a string	SUBSTRING('ABC',1,1)	A'

TRIM	Remove the longest string that contains specified characters from the left, right or both of the input string	TRIM(‘ABC ‘)	ABC’
UPPER	Convert a string to uppercase	UPPER(‘hI tHERE’)	HI THERE’
Split_part	splits a string on a specified delimiter and returns the nth substring. SPLIT_PART(string, delimiter, position)	Slipt_part(‘wel-come’,’-‘,2)	come

Date Functions

Function	Description	Example
current_date	Returns Current date	select current_date;
current_time	Returns Current Time	select current_time;
now()	Returns Current date and time	select now();
current_timesta mp	Returns Current date and time	select current_timestamp ;
extract(FIELD from date '<date>')	Return the FIELD part from the date. Valid FIELDS are below: YEAR, DAY, MONTH, HOUR, MINUTE, SECOND, MILLISECONDS, DOY, DOW, CENTURY, QUARTER	select extract(day from date '2021-05-24')
age (timestamp, ti mestamp)	Returns difference between given 2 dates or given date and now()	age(timestamp '2001-04-10', timestamp '1957-06-13')
pg_sleep(n)	Sleeps n number of seconds	select pg_sleep(10)
pg_sleep_for('interv al')	Sleeps for given interval	select pg_sleep('5 minutes')

Date Arithmetic

Date Addition

Date Additions can be done using interval key word.

```
select now() + interval '13 days'
select now() + interval '1 year'
select now() + interval '1 month'
select date '2021-02-15' + interval '1 month'
```

Interval can be in DAYS, MONTH, YEAR, HOURS, MINUTES and SECONDS

Date Subtraction

```
select date '2021-02-15' - date '2021-03-15' ;
select date '2021-02-15' + interval '30 days';
select ename, now()-hiredate from emp;
select ename, (now()-hiredate) from emp;
select ename, age(now(),hiredate) from emp;
```

Data Type Formatting Functions

```
to_char()
to_date()
to_number()
to_timestamp()
```

to_char() function can be used to format date or numeric values and output will be treated as string.

Examples:

```
to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS')
to_char(now(),  'YYYY-MON-DD')
to_char(125, '999') → 125
to_char(125.8::real, '999D9') → 125.8
to_char(-125.8, '999D99S') → 125.80-
```

to_date() Function

```
to_date('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05
to_date('05-Dec-2000', 'DD-MON-YYYY')  -> 2000-12-05
```

Date Time formatting Patterns

Pattern	Description
HH	hour of day (01–12)
HH12	hour of day (01–12)
HH24	hour of day (00–23)
MI	minute (00–59)
SS	second (00–59)

MS	millisecond (000–999)
AM, am, PM or pm	meridiem indicator (without periods)
YYYY	year (4 or more digits)
BC, bc, AD or ad	era indicator (without periods)
MONTH	full upper case month name (blank-padded to 9 chars)
MON	abbreviated upper case month name (3 chars in English, localized lengths vary)
MM	month number (01–12)
DAY	full upper case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars in English, localized lengths vary)
DD	day of month (01–31)
TZ	upper case time-zone abbreviation (only supported in to_char)

to_number() Function

```
to_number('122345', '99G999D9S')  ->  12,234.5
```

Numeric Formatting

Pattern	Description
9	digit position (can be dropped if insignificant)
0	digit position (will not be dropped, even if insignificant)
. (period)	decimal point
, (comma)	group (thousands) separator
PR	negative value in angle brackets
S	sign anchored to number (uses locale)
L	currency symbol (uses locale)
D	decimal point (uses locale)
G	group separator (uses locale)

RN	Roman numeral (input between 1 and 3999)
TH or th	ordinal number suffix
EEEE	exponent for scientific notation

Aggregate Functions

```
sum()
avg()
count()
min()
max()
string_agg()
```

Examples of Aggregate Function:

```
employees=# select sum(sal) from emp;
sum
----------
29025.00 (1 row)
employees=# select max(sal) from emp;
max
---------
5000.00 (1 row)
employees=# select min(sal) from emp;
min
--------
800.00 (1 row)
employees=# select count(sal) from emp;
count
-------
14 (1 row)
employees=# select avg(sal) from emp;
avg
-----------------------
2073.2142857142857143
(1 row)
employees=# select job, count(*) from emp group by job;
job | count
-----------+-------
CLERK | 4
PRESIDENT | 1
MANAGER | 3
SALESMAN | 4
ANALYST | 2
```

```
(5 rows)
employees=# select deptno, count(*) from emp group by deptno;
deptno | count
--------+-------
30| 6
10| 3
20| 5
(3 rows) employees=#
employees=# select deptno, count(*) from emp group by deptno having count(*) < 4;
deptno | count
--------+-------
10| 3
(1 row)
employees=#
employees=# select * from emp limit 3;
employees=# select * from emp order by ename;
employees=# select job, count(*) from emp group by job order by count(*);
job | count
-----------+-------
PRESIDENT | 1

job | count
-----------+-------
CLERK | 4
SALESMAN | 4
MANAGER | 3
ANALYST | 2
PRESIDENT | 1
(5 rows)
```

Comparison Operators

Operator	Description	Usage
<	Less than?	1 < 2
<=	Less than or equal to?	1 <= 2
<>	Not equal?	1 <> 2
=	Equal?	1 = 1
>	Greater than?	2 > 1
>=	Greater than or equal to?	2 >= 1
||	Concatenate strings	Postgre' || 'SQL'
~~	LIKE	scrappy,marc,hermit' ~~ '%scrappy%'
!~~	NOT LIKE	bruce' !~~ '%al%'
~	Match (regex), case sensitive	thomas' ~ '*.thomas*.'

~*	Match (regex), case insensitive	thomas' ~* '*.Thomas*.'
!~	Does not match (regex), case sensitive	thomas' !~ '*.Thomas*.'
!~*	Does not match (regex), case insensitive	thomas' !~ '*.vadim*.'
IS NULL	Test whether value is NULL	
IS NOT NULL	Negates NULL test	
IS TRUE	Test whether value is true	
IS DISTINCT	Test whether values are distinct	x IS DISTINCT y
LIKE	Pattern Matching	X LIKE '%s%'
NOT LIKE	Negates pattern Matching	
IN		X in (v1, v2,..vn)
NOT IN		
BETWEEN	Check if the given value is between given tange	
NOT BETWEEN	Negates the BETWEEN check	

Arithmetic Operators

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division (integer division truncates the result)	4 / 2	2
%	modulo (remainder)	5 % 4	1
^	exponentiation (associates left to right)	2.0 ^ 3.0	8
|/	square root	|/ 25.0	5
||/	cube root	||/ 27.0	3
@	absolute value	@ -5.0	5

PostgreSQL Advance GROUP BY

Postgres cab calculate Sub totals and grand totals using GROUP Operations using ROLLUP and CUBE clauses

ROLLUP:

ROLLUP clause can perform Grand totals with GROUP BY Clause

Example:

In this example we are calculating the sum of salary for each department and at end we are also calculating grand total for all departments.

```
SELECT
    d.dept_name,
    SUM(e.sal) AS total_salary
FROM
    emp e
JOIN
    dept d ON e.dept_id = d.dept_id
GROUP BY
    ROLLUP(d.dept_name);
```

CUBE Clause

CUBE clause can be used to calculate SUB-TOTALS and Grand TOTALs with GROUP BY Operations.

Example:

```
SELECT
    d.dname,
    e.ename,
    SUM(e.sal) AS total_salary
FROM
    emp e
JOIN
    dept d ON e.deptno = d.deptno
GROUP BY
    CUBE(d.dname, e.ename) order by d.dname;
```

PostgreSQL Joins, Unions & Sub Queries

Table joining technique allows to combine rows from from 2 or more table. Following join types can be used in PostgreSQL

- Cross Join
- Inner Join
- Natural Join
- Left Outer Join
- Right Outer Join
- Full Outer Join

Join Condition

The join condition determines which rows from the two source tables are considered to be a "match". The join condition is specified with the ON or USING clause or implicitly by the word NATURAL.

Cross Join

Cross Join is a cartesian product which contains all rows from first table are joined with all rows from second table. For example if the first table has m rows and second table has n rows then resultant cross joined tables will have m*n rows.

Inner Join

Inner also called as Join combines only matching rows for two tables and leaves out the rows without matches.

Syntax:

```
Select  <columns list>   from tableA A Inner Join  TableB  B  On  <condidition>;
Select  <column list>  from TableA A Join TableB B On <condition>;
Select  <column list>  from TableA A Join TableB B Using  <common column>;
```

Natural Join

Natural Join is Inner Join only difference is that there is no need to give any Join Condition.

```
Syntax:
Select  <column list>  from TableA A Natural Join TableB B;
```

Left Outer Join

Left Outer Join also called as Left Join combines all matching rows from both table plus rows from left side table which does not find matches also included.

Syntax:

```
Select  <columns list>   from tableA A Left Outer Join  TableB  B  On  <condidition>;
Select  <column list>  from TableA A Left Join TableB B On <condition>;
Select  <column list>  from TableA A Left Join TableB B Using  <common column>;
```

Right Outer Join

Right Outer Join also called as Right Join combines all matching rows from both table plus rows from Right side table which does not find matches also included.

Syntax:

```
Select  <columns list>   from tableA A Right Outer Join  TableB  B  On  <condidition>;
Select  <column list>  from TableA A Right Join TableB B On <condition>;
Select  <column list>  from TableA A Right Join TableB B Using  <common column>;
```

Full Outer Join

Full Outer Join also called as Full Join combines all matching rows from both table plus rows from both left side table and right side table which does not find matches also included.

Syntax

```
Select  <columns list>   from tableA A Full Outer Join  TableB  B  On  <condidition>;
Select  <column list>  from TableA A Full Join TableB B On <condition>;
Select  <column list>  from TableA A Full Join TableB B Using  <common column>;
```

Combining Queries (Unions)

Results of 2 or more queries can be combined using SET operations like UNION, INTERSECT. Supported set operations in PotgreSQL as follows:

- UNION
- UNION ALL
- INTERSECT
- EXCEPT

UNION

Union set operator allows to combine the results of 2 queries as a single result set.

The following are requirements for the queries in the syntax above:

- The number and the order of the columns must be the same in both queries.
- The data types of the corresponding columns must be the same or compatible.

Syntax:

```
select  <col1>, <col2>, ... from <table1>  UNION  select <col1>, <col2> ... from <table2>
```

UNION ALL

UNION ALL works same as UNION but it combines all elements from both the sets where as UNION returns common elements only once

Syntax:

```
select  <col1>, <col2>, ... from <table1>  UNION ALL  select <col1>, <col2> ... from
<table2>
```

INTERSECT

Intersect allows to combine results of 2 queries and returns only common rows between them.

Syntax:

```
select  <col1>, <col2>, ... from  <table1>  INTERSECT  select <col1>, <col2>,..... from
<table2>
```

EXCEPT

EXCEPT compares the result sets of two queries and returns the distinct rows from the first query that are not output by the second query.

Syntax:

```
select  <col1>, <col2>, ... from  <table1>  EXCEPT  select <col1>, <col2>,..... from
<table2>
```

Sub Queries

A Sub Query is also a query, which is defined under a main query. First Query is also called as outer query or main query and second query is also called as inner query or subquery.

Server first sub query executed based on the sub query value main query will get executed.

If a sub query send single value to its nearest main query then that sub query is called Single-Valued-Sub query.

If a sub query sends multiple values to its nearest main query then that sub query is called Multi-Valued-Sub query.

Note: If a sub query sends multiple values to its nearest main query then we have to use IN or NOT IN operator between Main query and Sub query.

Single valued Subquery:

```
SELECT * FROM TABLENAME WHERE COLUMNNAME = (SELECT STATEMENT WHERE CONDITION)
```

Multi valued subquery

A subquery which returns more than one row to the outer query is called as Multi Values subquery. Multivalues values sub queries can be handled by IN,

Multivalue SubQuery

```
select * from tablename where columname in (select statement where condition)
```

Example:

display all emp details whose earning salary above avg.salary

```
select * from emp where sal>=(select avg(sal) from emp)
select * from student where courseid in (select courseid from course where duration > 6)
```

Correlated Sub Query

In Correlated sub queries the outer query and the inner (sub) query are executed simultaneously.

Example:

display max salary for each dept and corresponding emp details

```
select * from emp e where sal=(select max(sal) from emp where deptno=e.deptno)
```

"First outer query gets executed, extracts 1 row at a time(candidate row) and that row is given to inner query for processing, inner query will provide its output to outer query and based on the condition outer query will display the extracted record".

Lesson 6 - Advance Data Types & Querying

Arrays

An array can store multiple values of same type. PostgreSQL allows declaring a column as array so that multiple values can be stored for that column for each row.

Array Declaration

```
CREATE TABLE <tablename> ( <normalcolumn>  <datatype>,  <arraycolumn_name>  <datatype>[]
... );

CREATE TABLE <tablename> ( <normalcolumn>  <datatype>,  <arraycolumn_name>
<datatype>[][] ... );
```

Example

```
create table arr_test (id int,  arr_col  int[]);
```

Inserting Values in Array Columns

Insert can be done along with other columns

Syntax

```
INSERT INTO  <tablename> VALUES (v1, '{pray elements separated by commas}' );
```

Example

```
INSERT INTO arr_test values (10, '{1000,2000,3000}');
```

Accessing Arrays

Arrays type columns can be assessed with the select statements as other datatypes.

Examples:

```
select * from arr_test;
select arr_col[2] from arr_test;
```

Updating an Array

Array columns can be updated with UPDATE query

Example

```
UPDATE arr_test SET arr_col = '{25000,25000,27000,27000}' WHERE id=10;
UPDATE sal_emp SET arr_col[2] = 5000 WHERE id=10;
```

Composite Types

A composite type represents the structure of a row or record; it is essentially just a list of field names and their data types.

Before using Composite Types in either as data type of column or on server side programing we need to create the composite types first.

Syntax:

```
CREATE TYPE <composite type name> AS (
    <col1>              <datatype>,
     <col2>              <datatype>,
     <col3>              <datatype>
);
```

Example:

```
CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

Inserting Values in Composite Types

```
Insert into <tablename>  values ('c1, '( val1 , val2 , ... )', c3...)
```

Example:

Lets Say table is

```
create table  weekly_stock (
  Weekofyear  int,
  item  inventory_item,
  year  decimal(4)
);


Insert into  weekly_stock values  (12, '("Coke", 1001, 134.00)', 2020);
```

Selecting Composite Types

select a particular column from composite type need the notation of (<composite type name>).<columnname>

```
Select * from weekly_stock;
Select (item).name from weekly_stock where item.supplier_id = 1001;
```

`()` must be used to enclose composite type columns when accessing

Updating Composite Types

```
update weekly_stock set item.name='PEPSI' where (item).supplier_id = 1001;
```

Hstore Columns

Hstore columns allows us to store data as a set of unique identifiers each of which have an associated value. This combination is also called as Key-Value-Pair. It is a useful technique to store semi-structured data in relational database.

Hstore is provided by CONTRIB module and the extension needs to be enabled for each database.

Enable Hstore Datatype

Connect to the required database and run the following command.

```
CREATE EXTENSION hstore;
```

Once the extension is created any table can have a column whose datatype is HSTORE.

Creating a Table with Hstore Column

Lets say we are collecting customers data which include customer addresses. A closer look at the addresses, you notice that Address consists of several key values pair.

For Example:

```
HouseNo   -->   134
Street  -->  James street,
Locality  -->  Jubilee Hills
city  --> Hyderabad
```

We can say that this data is in form of Key-Value-Pair. Let create a customer table with 3 column

```
cust_id, Custi_Name, Address
create table  customers ( cust_id  int,  cname  varchar(30), address  hstore);
```

Inserting Data in Hstore column

Insert statement can be used to data in hstore column.

Lets insert some values in the customer table created in previous topic.

```
insert into customers  values (10, 'Scott', ' "Hno" => "134", "screet => JamesStreet",
"locality => "Jubileehills", "city" => 'Hyderabad"');
```

Note: All the Key-Value-Pairs are enclosed in single quotes and individual KEYs or Values are enclosed in Double quotes and comma separated. Basically an Hstore column is a text column.

Querying an Hstore Column

Hstore columns can be queried using SELECT statement.

```
Select * from customers;
```

Query for a Specific Key

```
select name,  address -> 'city'  city from customers;
```

Using Key-Value-Pair in where clause

```
select  cname,  address ->  city  city from customers where  address -> 'locality' =
'JamesStreet';
```

Add New Key to existing Rows

```
update  customers set address = address || '"state"=>"Telangana"'::hstore;
```

It will update all rows in the table incase if more controlled update is needed then WHERE Clause can be used.

```
update  customers set address = address ||'"state"=>"Telangana"'::hstore where
cust_id=10;
```

Indexes on Hstore Column

Hstore columns support GIN or GiST indexes efficiently however HASH and B-Tree can also be used for equality checks only.

JSON/JSONB

JSON & JSONB datatype brings NoSQL support in PostgreSQL allowing users to save data in JSON objects. PostgreSQL validates the syntax any input and rejects if it does not confirm to JSON syntax.

JSON support was included in PostgreSQL in version 9.2 and JSONB support included in version 9.4.

JSONB is much better choice to use as it is faster and supports many features.

comparison JSON vs JSONB

JSON	**JSONB**
Stores data in plain text	Stores data in binary format
Preserve the indentation of input data	Does not preserve indentation of input data
Does not support indexing	Supports Indexing (GIN indexes)
Does not allow adding or removing field	Allows adding OR removing fields

Creating a Table With JSONB

```
create table emp_json (id serial, details JSONB);
```

Inserting Data in Emp_Json Table

```
insert into emp_json (details) values ('{ <valid JSON document> }'), ('{ <valid JSON
document> }'), ..
```

Querying

```
select * from emp_json;
```

Querying a Specific JSON Field

```
select details -> 'ename' from test;
```

Avoiding Quotes in Output

```
select details ->> 'ename' from test;
```

Using a JSON Field in WHERE Clause

```
select details ->> 'ename', details ->> 'job', details ->> 'sal' from test where details
->> 'ename' = 'FORD';
select details ->> 'ename' as ename, details ->> 'job' as Job, details ->> 'sal' as sal
from test where cast (details ->> 'sal' as integer) >= 2000;
```

Using a JSON Field in Order by Clause

```
select details ->> 'ename' as ename, details ->> 'job' as Job, details ->> 'sal' as sal
from test order by cast (details ->> 'sal' as numeric);
```

Using a JSON Field in Group by Clause

```
select details ->> 'job' as Job, count(*) from test group by details ->> 'job'
```

Check if the JSON Field Exists

```
select details ->> 'ename' as ename, details ->> 'job' as Job, details ->> 'sal' as sal
from test where details ? 'sal';
```

Check if a Key Value Pair Exists in

```
select details ->> 'ename' as ename, details ->> 'job' as Job, details ->> 'sal' as sal
from emp_json where details @> '{"ename":"KING"}';
```

Updating a Field (Only JSONB)

```
update emp_json set details = jsonb_set(details, '{sal}', '1000',false) where details ->>
'ename' = 'SMITH';
```

Removing a Field (Only JSONB)

```
update emp_json set details = details - 'comm' where details ->> 'ename' = 'SMITH';
```

Range Data Types

Range types are data types representing a range of values of some element type like timestamp, int etc.

PostgreSQL comes with the following built-in range types:

- `int4range` — Range of integer, int4multirange — corresponding Multirange
- `int8range` — Range of bigint, int8multirange — corresponding Multirange
- `numrange` — Range of numeric, nummultirange — corresponding Multirange
- `tsrange` — Range of timestamp without time zone, tsmultirange — corresponding Multirange
- `tstzrange` — Range of timestamp with time zone, tstzmultirange — corresponding Multirange
- `daterange` — Range of date, datemultirange — corresponding Multirange

Range Operators

Operator	Description	Example	Result
=	**equal**	int4range(1,5) = '[1,4]'::int4range	t
<>	**not equal**	numrange(1.1,2.2) <> numrange(1.1,2.3)	t
<	**less than**	int4range(1,10) < int4range(2,3)	t
>	**greater than**	int4range(1,10) > int4range(1,5)	t
<=	**less than or equal**	numrange(1.1,2.2) <= numrange(1.1,2.2)	t
>=	**greater than or equal**	numrange(1.1,2.2) >= numrange(1.1,2.0)	t
@>	**contains range**	int4range(2,4) @> int4range(2,3)	t
@>	**contains element**	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp	t
<@	**range is contained by**	int4range(2,4) <@ int4range(1,7)	t
<@	**element is contained by**	42 <@ int4range(1,7)	f
&&	**overlap (have points in common)**	int8range(3,7) && int8range(4,12)	t
<<	**strictly left of**	int8range(1,10) << int8range(100,110)	t
>>	**strictly right of**	int8range(50,60) >> int8range(20,30)	t
&<	**does not extend to the right of**	int8range(1,20) &< int8range(18,20)	t
&>	**does not extend to the left of**	int8range(7,20) &> int8range(5,10)	t
-|-	**is adjacent to**	numrange(1.1,2.2) -|- numrange(2.2,3.3)	t
+	**union**	numrange(5,15) + numrange(10,20)	[5,20)
*	**intersection**	int8range(5,15) * int8range(10,20)	[10,15)

Operator	Description	Example	Result
-	**difference**	int8range(5,15) - int8range(10,20)	[5,10)

Range Functions

Function	Return Type	Description	Example	Result
lower(anyrange)	**range's element type**	**lower bound of range**	lower(numrange(1.1,2.2))	1.1
upper(anyrange)	**range's element type**	**upper bound of range**	upper(numrange(1.1,2.2))	2.2
isempty(anyrange)	**boolean**	**is the range empty?**	isempty(numrange(1.1,2.2))	false
lower_inc(anyrange)	**boolean**	**is the lower bound inclusive?**	lower_inc(numrange(1.1,2.2))	true
upper_inc(anyrange)	**boolean**	**is the upper bound inclusive?**	upper_inc(numrange(1.1,2.2))	false
lower_inf(anyrange)	**boolean**	**is the lower bound infinite?**	lower_inf('(,)'::daterange)	true
upper_inf(anyrange)	**boolean**	**is the upper bound infinite?**	upper_inf('(,)'::daterange)	true

Examples:

```
CREATE TABLE tennis_court (court_id int, reserved tsrange);
```

Insert data:

```
INSERT INTO tennis_court VALUES (10, '[2024-11-25 8:00, 2024-11-25 10:00]');
INSERT INTO tennis_court VALUES (11, '[2024-11-25 8:00, 2024-11-25 9:30]');
```

Check if the court 10 is booked between 9:30 to 11:30 on 2024-11-25

```
select * from tennis_court where reserved && tsrange('2024-11-25 09:30', '2024-11-25
11:30') and court_id=10;
 court_id |                     reserved
----------+-----------------------------------------------
       10 | ["2024-11-25 08:00:00","2024-11-25 10:00:00"]
```

CTE - Common Table Expressions

Common Table Expressions (CTE) are useful to simplify the complex queries. CTEs are the auxiliary query whose results can be used by the primary query. Result set of CTEs auxiliary query is temporary until primary query execution completes.

Syntax:

```
WITH <cte_name> AS (auxiliary query) primary_query;
```

Example:

```
WITH emp_salgrade AS (select ename, job, sal, deptno, grade from emp join salgrade on sal
between losal and hisal) select ename, job, sal, grade from emp_salgrade es join dept d
on es.deptno=d.deptno;
```

Data Modifying Statement

Both the primary and auxiliary queries can be data modifying queries like INSERT, UPDATE or DELETE. Temporary result set of the auxiliary query can be referenced by primary query only if the RETURNING * clause is used otherwise not.

Example:

```
WITH del_table AS (delete from visitor_logs where log_date < CURRENT_DATE - interval '1
month' RETURNING *) insert into visitor_archive select * from del_table;
```

The above statement deletes the rows from visitor_log table and also insert them in visitor_archive table.

Window Functions

Windows functions performs the calculations on set of rows that are related to current row.

Window Functions

- min(),
- max(),
- avg()
- rank() – ranks the output
- row_number() - generates the row numbers
- lag() - Showing the value of previous row in current row
- lead() - Showing the value of next row in current row
- last_value() - Showing last value
- first_value() - Showing first value

min()

```
select ename, job, sal, min(sal) over (partition by job) from emp;
select ename, job, sal, min(sal) over (partition by job order by sal desc ) from emp
select ename, job, sal, min(sal) over () from emp
```

rank()

```
select ename, job, sal, rank() over (order by sal desc ) from emp;
select ename, job, sal, rank() over (partition by job order by sal desc ) from emp;
```

row_number()

```
select ename, job, sal, row_number() over (order by sal desc ) from emp;
```

lag()

```
select ename, job, sal, lag(sal,1) over (partition by job order by sal desc ) as prev_sal
from emp;
```

lead()

```
select ename, job, sal, lead(sal,1) over (partition by job order by sal desc ) as
next_sal from emp;
```

last_value()

```
select ename, job, sal, last_value(sal) over (partition by job order by sal desc ) from
emp;
```

first_value()

```
select ename, job, sal, first_value(sal) over (partition by job order by sal desc ) from
emp;
```

Full Text Search

PostgreSQL has text search operators like ~, ~*, LIKE, and ILIKE but they lack the Linguistic support, provide ranking for search result based on which result is more relevant to the search query and lack index support. To over come these issues and provide more flexible linguistic search PostgreSQL has provided Text Search capabilities.

Full Text Search terminology

Document : Document is a search unit and it is a text body like a magazine article, email etc.. Usually a TEXT column from a table.
tsvector: A data type that stores preprocessed documents in a format optimized for searching.
tsquery: A data type used to represent search queries.
@@ (match operator) : This operator is used to match the Document (tsvector) with search query (tsquery). Returns true or false depending if the match is found or not.

Text Analysis

Tokenization: Breaking down text into tokens (words).
Normalization: Converting tokens into lexemes (normalized words) to handle different forms of the same word.

Stop Words: Common words that are ignored during searches (e.g., "the", "and").

tsquery operators

Operator	Description	tsquery Example
&	AND operator	‘cat & fat’
|	OR Operator	‘cat | rat’
!	NOT operator	‘!rat’
<->	FOLLOWED BY	‘fat <-> cat’
()	Parenthesis to group search terms	‘(fat <-> cat) & hat’

FTS Functions

Function	Description	Example
to_tsvector(<config>, <doc>)	Converts a text into searcheable vector	Select to_tsvector(english, ‚sam the fat cat‘);
To_tsquery(<config>, <search query>	Converts words to tsquery	Select to_tsquery(english, ‚sam & fat‘)
tsvector_to_array (tsvector)	Converts tsvector to an array consisting of lexemes	Tavector_to_array(comments)
array_to_tsvector (text[])	Converts an array to tsvector	
plainto_tsquery ([config regconfig,] query text)	Converts plain text to tsquery	
phraseto_tsquery ([config regconfig,] query text)	Converts text to tsquery ignoring punctuations	

Search Configuration

Following configuration itemscan used for FTS

default_text_search_config: This parameter specifies the default text search configuration to be used by text search functions if an explicit configuration is not provided. It can be set in the postgresql.conf file or for an individual session using the SET command.

dictionaries: PostgreSQL supports various dictionaries like simple, synonym, thesaurus, ispell, and snowball. These dictionaries help in processing and normalizing text during searches.

parsers: The text search parser is responsible for breaking down text into tokens. You can specify a parser when creating a text search configuration.

stop words: These are common words that are usually ignored during searches. You can define stop words in your dictionaries to improve search efficiency

Search Examples:

Basic text search can be performed by matching tsvector and tsquery data types

Example1:

```
select 'Sam the fat cat with a hat can see the Jam'::tsvector @@ 'cat & hat'::tsquery;
 ?column?
----------
 t
```

Example2

```
select 'cat & hat' @@ 'Sam the fat cat with a hat can see the Jam'::tsvector;
 ?column?
----------
 t
(1 row)
```

Example3

```
select 'Sam the fat cat with a hat can see the Jam'::tsvector @@ 'cat & rat'::tsquery
;
 ?column?
----------
 f
(1 row)
```

Above match failed because there is no ‘rat’ in document.

Example4

```
select 'Sam the fat cat with a hat can see the Jam'::tsvector @@ '!rat'::tsquery;
 ?column?
----------
 t
(1 row)
```

Example5

```
select 'cat ran the running test'::tsvector @@ 'run'::tsquery;
 ?column?
----------
 f
(1 row)
```

Example6

```
select to_tsvector('cat ran the running test')  @@ to_tsquery('run');
 ?column?
----------
t
(1 row)
```

Creating a table with tsvector Column

To perform FT Search on table columns, it is recommended to have a tsvector column in table rather using `to_tsvector()` function for every search. This also allows to create a index for FTS.

Example:

```
CREATE TABLE user_feedback (id INT PRIMARY KEY, fb_date TIMESTAMP, rating INT CHECK
(rating between 1 and 5), comments  TEXT,  ft_search  TSVECTOR GENERATED ALWAYS
to_tsvector(comments) stored);
```

Indexes of tsvector Columns

GIN indexes are preferred on tsvector column however GiST indexes also can be used.

Lesson 7 - Views and Temp Tables

PostgreSQL Views

The view is not physically object or table. Instead, the query is run every time the view is referenced in a query.

PostgreSQL supports 2 types Views

- Normal Views
- Materialized Views

Normal Views

Creating Views

Before a view is used it needs to be created.

syntax

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] VIEW name [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Syntax Explanation

TEMPORARY or TEMP

If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session.

name

The name (optionally schema-qualified) of a view to be created

WITH (options)

This clause specifies optional parameters for a view; the following parameters are supported:

`check_option (string)`

allowed values are LOCAL or CASCADED.

Local: New rows are only checked against the conditions defined directly in the view itself

CASCADED: New rows are checked against the conditions of the view and all underlying base views

`security_barrier (boolean)`

This should be used if the view is intended to provide row-level security.

`security_invoker (boolean)`

This option causes the underlying base relations to be checked against the privileges of the user of the view rather than the view owner

query

A SELECT command which will provide the columns and rows of the view.

WITH [CASCADED | LOCAL] CHECK OPTION

When this option is specified, INSERT and UPDATE commands on the view will be checked to ensure that new rows satisfy the view-defining condition If they are not, the update will be rejected.

This same as WITH (options) clause

Changing the Query of View

It is possible to change the query of the view only if all the existing columns are included in new query otherwise such change is rejected however where clause can be changed.

To change the query of any view use

```
CREATE OR REPLACE VIEW name    AS New-Query;
```

Altering View

A view can be alerted to change OWNER, SCHEMA, RENAME and view options

Syntax

```
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER | SESSION_USER };
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name;
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema;
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] );
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] );
```

Drop View

A view can be dropped when it is not referenced by other views otherwise a error is returned.

Syntax:

```
DROP VIEW  name [CASCADE];
```

CASCADE option deletes all dependent objects as well

Materialized Views

Materialized views are created same like normal view the only difference is that the results are persisted in materialized until materialized is refreshed. Materialized view behaves similar to tables.

Materialized view are read only.

Creating Materialized View

Syntax:

```
CREATE MATERIALIZED VIEW name AS query;
```

Example

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM emp;
```

Refreshing the Materialized view

Changes made to the base table are not reflected in materialized view until it is refreshed.

Syntax

```
REFRESH MATERIALIZED VIEW view name;
```

Example

```
REFRESH MATERIALIZED VIEW mymatview;
```

Indexes

Indexes can be created on materialized views in same way as they can be created on tables

Dropping Materialized View

A materialized view can be dropped just like any other object

Syntax

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name;
```

Example

```
DROP MATERIALIZED VIEW myview;
```

Temp Tables

PostgreSQL allows a session to create a temporary table which is visible for this session and will be dropped immediately along with its indexes at the end of the session.

Each session requiring temp table must create by itself. This allows the sessions to use the temporary table for different purposes.

Autovaccum doesn't touch temp tables however if stats needs to updated then ANALYZE command can be used.

Syntax:

```
CREATE TEMPORARY TABLE temp_table_name( … ) [ON COMMIT DELETE ROWS | DROP];
```

Indexes in PostgreSQL

An index allows the database server to find and retrieve specific rows much faster than it could do without an index.

Types of Indexes Supported by PostgreSQL

- Hash Indexes
- B-Tree Indexes
- GIN Indexes
- GiST Indexes
- BRIN Indexes

B-Tree Indexes

This the default type of index created in PostgreSQL. B-Tree indexes can handle equality searches and range searches efficiently. If the searches with following operators are done then optimizer selects B-Tree index, if present.

Operations supported by B-tree

```
=, >, <, >=, <=, IS NULL, IS NOT NULL, BETWEEN, IN
```

However **LIKE** operator is supported only if the search term is anchored at the beginning of string (eg. **ename LIKE "foo%**).

Syntax:

```
CREATE INDEX <indexname>  ON <table_name> (<indexed_column> [ASC | DESC] [NULLS {FIRST |
LAST }, ,indexed_column2....);
```

Indexes and ORDER BY

You can adjust the ordering of a B-tree index by including the options ASC, DESC, NULLS FIRST, and/or NULLS LAST when creating the index;

Syntax:

```
CREATE INDEX <index name> ON <tablename> (<col> NULLS FIRST);
CREATE INDEX <index name> ON <tablename>  (<col> DESC NULLS LAST);
```

This will save sorting time spent by the query

Multicolumn Index

If any of frequently running queries uses multiple columns in where clause then for such queries a multicolumn index can be created which includes all columns used in where clause.

This speeds up the query execution.

Syntax:

```
CREATE INDEX test_idx1 ON test (id_1, id_2);
```

Currently, only the B-tree, GiST, GIN, and BRIN index types support multicolumn indexes. Up to 32 columns can be specified

Unique Indexes

Indexes can also be used to enforce uniqueness of a column's value or the uniqueness of the combined values of more than one column

Syntax:

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

Currently only B-tree indexes can be declared unique.

PostgreSQL automatically creates a unique index when a unique constraint or primary key is defined for a table

Functional Indexes

An index can be created on a computed value from the table columns. Usually if the expression used in where clause optimizer ignore the indexes and prefer full tablescan. We can avoid full tablescans by creating the index with expression instead of column.

Syntax:

```
CREATE INDEX <index name>  ON <tablename>  (lower(col1));
```

They can also be created on user defined functions

Partial Index

A partial index is an index built over a subset of a table rather than whole table which reduces maintenance of the index.

The index contains entries only for those table rows that satisfy the predicate and also optimizer uses this index when the query predicate matches the index predicate.

Syntax:

```
CREATE INDEX <indexname> on <tablename> (<column>) where <condition;
```

Hash Indexes

Hash Indexes are useful in speeding up queries which does simple equality comparison.

Syntax:

```
CREATE INDEX <indexname>  ON <table_name>  USING HASH (<indexed_column> );
```

GiST Indexes

The GiST, or Generalized Search Tree index type indexes are useful for searching Geometric data or Full-Text searches. They are particularly helpful in Geometric data searches.

Syntax:

```
CREATE INDEX <indexname>  ON <table_name>  USING GSIT (<indexed_column>);
```

GIN Indexes

Generalized Inverted Index (GIN) is an index type that is beneficial when a datatype has multiple values within a single column like TEXT columns or Arrays. GIN Indexes are mostly used with Full-Text searches.

Syntax:

```
CREATE INDEX <indexname>  ON <table_name>  USING GIN (<indexed_column>);
```

BRIN Indexes

Binary Range Indexes (BRIN) are particularly effective aginst large data sets, data types with linear search order like integers or dates.

BRIN indexes are less expensive than B-Tree indexes.

Syntax:

```
CREATE INDEX <indexname>  ON <table_name>  USING BRIN (<indexed_column>);
```

Index-Only Scans and Covering Indexes

When all required column are available index then postgreSQL returns the query result from index only avoiding reading heap data (table data file) this speeds up the query execution.

It it is recommended to create multicolumn indexes for those frequently running queries which uses only few columns in select.

B-Tree has complete support for index only scans but GIST / GIN has limited support.

Lesson 8 - PostgreSQL MVCC and Locks

PostgreSQL supports Multi Versioning and Concurrency Control (MVCC) by providing Isolation Levels, Transactions and Locks.

Transactions

PostgreSQL supports multi statement DML ACID compliant transactions and all other data changing statements are also atomic. However for multi statement DML transactions session needs start the transaction explicitly using START TRANSACTION statement otherwise PostgreSQL do a auto commit separately after each DML statement.

Syntax:

```
start transaction;
  .....
  .....
  Some DML statements
COMMIT / ROLLBACK;
```

Every transaction ends with a COMMIT or ROLLBACK.

Example:

```
start transaction;
update emp set sal = sal + 100;
select * from emp;
rollback;
```

ACID Properties

PostgreSQL supports ACID compliant transaction ensuring the relieability of the database

PostgreSQL Isolation Levels

PostgreSQL supports all standard SQL isolation levels

- Read Uncommitted
- Read Committed
- Repeatable Read
- Serializable

Default Isolation level is read committed.

In postgres repeatable read isolation level is implemented it in a way that the phantom read problem is also taken care of.

Check Current Isolation Level

To know the current isolation level configured in postgres we can examine the server configuration parameter DEFAULT_TRANSACTION_ISOLATION

Example

```
show default_transaction_isolation;
```

Changing Default Isolation Level

Any time default isolation level can be changed by running the following command

Example:

```
ALTER SYSTEM SET DEFAULT_TRANSACTION_ISOLATION = 'serializable';
```

and reload PostgreSQL Service

Start transaction with a particular Isolation level

In PostgreSQL any transaction can be started with a particular isolation level using START TRANSACTION command

Syntax:

```
START TRANSACTION [
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }]
  .....
  Some DML statements
COMMIT / ROLLBACK;
```

Example:

```
start transaction isolation level repeatable read;
update emp set sal = sal + 100;
select * from emp;
rollback;
```

PostgreSQL Locks

Various lock modes are provided to the applications / users to control MVCC behavior during the transactions.

Most of PostgreSQL commands automatically acquire locks of appropriate levels & modes to ensure that referenced tables / rows are not dropped or modified in incompatible ways while the command executes

Table Level Locking.

Most of the table level locks are acquired implicitly when certain data changing statements are executed like ALTER, TRUNCATE, UPDATE etc.

Explicit Table Level Locks

PostgreSQL also allows us to lock the tables explicitly using LOCK command.

LOCK TABLE command works only within a transaction block and results in error if used outside a transaction block.

ALL locks are released when the transaction holding them ends.

Syntax:

```
  LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]
```

where lockmode is one of:

```
ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE| SHARE | SHARE ROW
EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Parameters:

name

The name (optionally schema-qualified) of an existing table to lock. If ONLY is specified before the table name, only that table is locked. If ONLY is not specified, the table and all its descendant tables (if any) are locked. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

The command LOCK TABLE a, b; is equivalent to LOCK TABLE a; LOCK TABLE b;. The tables are locked one-by-one in the order specified in the LOCK TABLE command.

lockmode

The lock mode specifies which locks this lock conflicts with.

If no lock mode is specified, then ACCESS EXCLUSIVE, the most restrictive mode, is used.

NOWAIT

Specifies that LOCK TABLE should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

Table-level Lock Modes

ACCESS SHARE

SELECT statement acquires this lock implicitly on all tables referenced in query.

ROW SHARE

The SELECT FOR UPDATE and SELECT FOR SHARE commands acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).

ROW EXCLUSIVE

In general, this lock mode will be acquired by any command that modifies data in a table.

The commands UPDATE, DELETE, and INSERT acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables).

SHARE UPDATE EXCLUSIVE

This mode protects a table against concurrent schema changes and VACUUM runs.

Acquired implicitly by VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY, CREATE STATISTICS and ALTER TABLE VALIDATE and other ALTER TABLE variants (for full details see ALTER TABLE).

SHARE

This mode protects a table against concurrent data changes.

Acquired implicitly by CREATE INDEX (without CONCURRENTLY).

SHARE ROW EXCLUSIVE

This mode protects a table against concurrent data changes, and is self-exclusive so that only one session can hold it at a time.

Acquired implicitly by CREATE COLLATION, CREATE TRIGGER, and many forms of ALTER TABLE (see ALTER TABLE).

EXCLUSIVE

This mode allows only concurrent ACCESS SHARE locks, i.e., other transactions can perform reads only from the table in parallel with a transaction holding this lock mode.

Acquired implicitly by REFRESH MATERIALIZED VIEW CONCURRENTLY.

ACCESS EXCLUSIVE

This mode guarantees that the holder is the only transaction accessing the table in any way.

Acquired implicitly by the DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL, and REFRESH MATERIALIZED VIEW (without CONCURRENTLY) commands. Many forms of ALTER TABLE also acquire a lock at this level. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Table Level Lock compatibility chart.

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X

ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

Row Level Locks

Row-level locks do not affect data querying; they prevent other transactions from writing to the locked rows.

Row Level locks are released when a transaction completes.

There are 4 ROW level locks provided.

These locks can be aquired explicitly by using select ... <LOCK MODE>;

- FOR SHARE
- FOR UPDATE
- FOR KEY SHARE
- FOR NO KEY UPDATE

FOR UPDATE

FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents them from being locked, modified or deleted by other transactions until the current transaction ends.

Within a REPEATABLE READ or SERIALIZABLE transaction, however, an error will be thrown if a row to be locked has changed since the transaction started.

FOR SHARE

Behaves similarly to FOR NO KEY UPDATE, except that it acquires a shared lock rather than exclusive lock on each retrieved row. A shared lock blocks other transactions from performing UPDATE, DELETE, SELECT FOR UPDATE or SELECT FOR NO KEY UPDATE on these rows, but it does not prevent them from performing SELECT FOR SHARE or SELECT FOR KEY SHARE.

FOR NO KEY UPDATE

Behaves similarly to FOR UPDATE, except that the lock acquired is weaker: this lock will not block SELECT FOR KEY SHARE commands that attempt to acquire a lock on the same rows. This lock mode is also acquired by any UPDATE that does not acquire a FOR UPDATE lock.

FOR KEY SHARE

Behaves similarly to FOR SHARE, except that the lock is weaker: SELECT FOR UPDATE is blocked, but not SELECT FOR NO KEY UPDATE. A key-shared lock blocks other transactions from performing DELETE or any UPDATE that changes the key values, but not other UPDATE, and neither does it prevent SELECT FOR NO KEY UPDATE, SELECT FOR SHARE, or SELECT FOR KEY SHARE.

Lock Mode Compatibility Chart below

Requested Lock Mode	Current Lock Mode

	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

Explicitly Acquiring the Row Level Locks

Syntax:

```
Select  .......   <Lock mode>;
```

Example:

```
select * from where deptno = 20 for update
```

Lock Monitoring

One of the frequent causes of performance issues is database locks. A DBA must monitor locks to identify queries causing them and optimizing those queries.

Tables and functions useful to monitoring and troubleshooting database locks:

pg_stat_activity view:

A view with one entry per server process (user session), showing details of the running query for each.

pg_locks view:

Information on current locks held within the database by open transactions, with one row per lockable object.

pg_blocking_pids():

A function that can find the process IDs (PIDs) of sessions that are blocking the PostgreSQL server process of a supplied PID.

pg_cancel_backend():

Function that cancels the currently running query by sending a SIGINT to a process ID.

pg_terminate_backend():

Terminate a backend process completely (the query and usually the connection) on the database (uses SIGTERM instead of SIGINT).

Details of pg_stat_activity

This view provides the details of current session activity on the server. From this view we can track all the current session and current transactions.

Column Type Description
datid oid OID of the database this backend is connected to
datname name Name of the database this backend is connected to
pid integer Process ID of this backend
leader_pid integer Process ID of the parallel group leader, if this process is a parallel query worker. NULL if this process is a parallel group leader or does not participate in parallel query.
usesysid oid OID of the user logged into this backend
usename name Name of the user logged into this backend
application_name text Name of the application that is connected to this backend
client_addr inet IP address of the client connected to this backend. If this field is null, it indicates either that the client is connected via a Unix socket on the server machine or that this is an internal process such as autovacuum.
client_hostname text Host name of the connected client, as reported by a reverse DNS lookup of client_addr. This field will only be non-null for IP connections, and only when **log_hostname** is enabled.
client_port integer TCP port number that the client is using for communication with this backend, or -1 if a Unix socket is used. If this field is null, it indicates that this is an internal server process.

backend_start timestamp with time zone Time when this process was started. For client backends, this is the time the client connected to the server.
xact_start timestamp with time zone Time when this process' current transaction was started, or null if no transaction is active. If the current query is the first of its transaction, this column is equal to the query_start column.
query_start timestamp with time zone Time when the currently active query was started, or if state is not active, when the last query was started
state_change timestamp with time zone Time when the state was last changed
wait_event_type text The type of event for which the backend is waiting, if any; otherwise NULL.
wait_event text Wait event name if backend is currently waiting, otherwise NULL.
state text Current overall state of this backend. Possible values are: • active: The backend is executing a query. • idle: The backend is waiting for a new client command. • idle in transaction: The backend is in a transaction, but is not currently executing a query. • idle in transaction (aborted): This state is similar to idle in transaction, except one of the statements • in the transaction caused an error. • fastpath function call: The backend is executing a fast-path function. • disabled: This state is reported if **track_activities** is disabled in this backend.
backend_xid xid Top-level transaction identifier of this backend, if any.
backend_xmin xid The current backend's xmin horizon.

query text Text of this backend's most recent query. If state is active this field shows the currently executing query. In all other states, it shows the last query that was executed. By default the query text is truncated at 1024 bytes; this value can be changed via the parameter **track_activity_query_size**.
backend_type text Type of current backend. Possible types are autovacuum launcher, autovacuum worker, logical replication launcher, logical replication worker, parallel worker, background writer, client backend, checkpointer, startup, walreceiver, walsender and walwriter. In addition, background workers registered by extensions may have additional types.

Details of pg_lock view

Provides information about the currently granted locks and currently waiting locks.

Column Type Description
locktype text Type of the lockable object: relation, extend, frozenid, page, tuple, transactionid, virtualxid, spectoken, object, userlock, or advisory.
database oid (references pg_database.oid) OID of the database in which the lock target exists, or zero if the target is a shared object, or null if the target is a transaction ID
relation oid (references pg_class.oid) OID of the relation targeted by the lock, or null if the target is not a relation or part of a relation
page int4 Page number targeted by the lock within the relation, or null if the target is not a relation page or tuple
tuple int2 Tuple number targeted by the lock within the page, or null if the target is not a tuple
virtualxid text Virtual ID of the transaction targeted by the lock, or null if the target is not a virtual transaction ID

transactionid xid ID of the transaction targeted by the lock, or null if the target is not a transaction ID
classid oid (references pg_class.oid) OID of the system catalog containing the lock target, or null if the target is not a general database object
objid oid (references any OID column) OID of the lock target within its system catalog, or null if the target is not a general database object
objsubid int2 Column number targeted by the lock (the classid and objid refer to the table itself), or zero if the target is some other general database object, or null if the target is not a general database object
virtualtransaction text Virtual ID of the transaction that is holding or awaiting this lock
pid int4 Process ID of the server process holding or awaiting this lock, or null if the lock is held by a prepared transaction
mode text Name of the lock mode held or desired by this process
granted bool True if lock is held, false if lock is awaited
fastpath bool True if lock was taken via fast path, false if taken via main lock table

Useful Queries to monitor and troubleshoot locking issues

Find queries running more than 2 minutes.

```
SELECT pid, user, query_start, (now() - query_start) AS query_time, query, state,
wait_event_type, wait_event FROM pg_stat_activity WHERE (now() - query_start) > interval
'2 minutes';
```

Blocking and blocked queries

Following query list out the blocking query, blocked query, duration.

```
SELECT COALESCE(blockingl.relation::regclass::text,blockingl.locktype) as locked_item,
now() - blockeda.query_start AS waiting_duration, blockeda.pid AS blocked_pid,
blockeda.query as blocked_query, blockedl.mode as blocked_mode, blockinga.pid AS
```

```
blocking_pid, blockinga.query as blocking_query, blockingl.mode as blocking_mode FROM
pg_catalog.pg_locks blockedl JOIN pg_stat_activity blockeda ON blockedl.pid =
blockeda.pid JOIN pg_catalog.pg_locks blockingl  ON((
(blockingl.transactionid=blockedl.transactionid) OR (blockingl.relation=blockedl.relation
AND blockingl.locktype=blockedl.locktype)) AND blockedl.pid != blockingl.pid) JOIN
pg_stat_activity blockinga ON blockingl.pid = blockinga.pid AND blockinga.datid =
blockeda.datid WHERE NOT blockedl.granted AND blockinga.datname = current_database();
```

Blocked query and backend pid details

```
select pid,  usename, pg_blocking_pids(pid) as blocked_by,  query as blocked_query from
pg_stat_activity where cardinality(pg_blocking_pids(pid)) > 0;
```

Lesson 9 - Table Partitioning

In PotgreSQL table partitioning is implemented as parent table and child tables where a parent table is partitioned table and child tables are individual partitions. Child tables behaves independently and can be queried independently as well.

Any dml operations on partitioned table (parent table) are actually performed on child table. Basically partitioned table remains empty.

From version 10 postgres is supporting declarative table partitioning otherwise in prior versions table partitioning was achieved by creating parent and child tables and then creating triggers on parent table to make sure all DML operations occurs on child table.

PostgreSQL offers built-in support for the following forms of partitioning:

Range Partitioning

The table is partitioned into "ranges" defined by a key column or set of columns.

List Partitioning

The table is partitioned by explicitly listing which key values appear in each partition.

Hash Partitioning

Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder.

Range Partition

Syntax

```
CREATE TABLE <parent table>   ( ............   )    PARTITION BY RANGE (part_column);
CREATE TABLE <child table1 name>  PARTITION OF <parent table>  FOR VALUES FROM (MINVALUE)
TO v1;
CREATE TABLE <child table2 name>  PARTITION OF <parent table>  FOR VALUES FROM (v2) TO
(v3);
CREATE TABLE <child table3 name>  PARTITION OF <parent table>  FOR VALUES FROM (v4) TO
(MAXVALUE);
```

Example:

- create master table

```
    create table emp (id int, name varchar(10), salary money) partition by range
(salary);
```

- create child table (partitions)

```
  create table p1 PARTITION OF emp FOR VALUES FROM (MINVALUE) to (5000);
  create table p2 PARTITION OF emp FOR VALUES FROM (5000) to (10000);
  create table p3 PARTITION OF emp FOR VALUES FROM (10000) to (MAXVALUE);
```

List Partition

Syntax

```
CREATE TABLE <parent table>  (.............)  PARTITION BY LIST(<column>);
CREATE TABLE <child table1> PARTITION OF <parent table>  FOR VALUES IN (l1,l2,...);
CREATE TABLE <child table1> PARTITION OF <parent table>  FOR VALUES IN (l1,l2,...);
CREATE TABLE <child table1> PARTITION OF <parent table> DEFAULT;
```

Example

- create master table

```
create table products (id int, description varchar(10), category varchar(20), stock int,
price decimal(12,2)) partition by list (category);
```

- create child table (partitions)

```
  create table p1 PARTITION OF products FOR VALUES IN  ('GROCERY','FRESHOS');
  create table p2 PARTITION OF emp FOR VALUES IN ('CLOTHING');
  create table p3 PARTITION OF emp FOR VALUES IN ('HOMEELEC','MOBILES');
  create table p3 PARTITION OF emp DEFAULT;
```

Hash Partition

Example

```
CREATE TABLE new_table ( id uuid, name varchar(30) ) PARTITION BY HASH (id);
CREATE TABLE new_table_0 PARTITION OF new_table FOR VALUES WITH (MODULUS 3, REMAINDER 0);
CREATE TABLE new_table_1 PARTITION OF new_table FOR VALUES WITH (MODULUS 3, REMAINDER 1);
CREATE TABLE new_table_2 PARTITION OF new_table FOR VALUES WITH (MODULUS 3, REMAINDER 2);
```

Foreign Key Support

Partitioned tables have limited foreign key support i,e a partitioned table can be a child table but cannot be a parent table in referential integrity.

Partition Management

PostgreSQL allows adding new partitions, removing partitions and attaching a existing table as partition and detaching a partition from portioned table

Adding a Partition

You can add new partitions as needed.

Example:

```
CREATE TABLE new_table_3 PARTITION OF new_table FOR VALUES WITH (MODULUS 3, REMAINDER 3);
```

Deleting Partition

Deleting a partition is nothing but dropping the table representing that partition

Example:

```
DROP TABLE new_table_3;
```

Attaching a Partition

ATTACH PARTITION is useful when you have a new table with data that you want to include in the partitioned table.

Example:

```
ALTER TABLE products ATTACH PARTITION p4 FOR VALUES FROM (10001) TO (15000);
```

Detaching a Partition

Detaching partition is nothing but removing the sub-table representing the given partition from the Partitioned Table. Once removed detached table behave like normal table.

Example:

```
ALTER TABLE products DEATTACH PARTITION p4 FOR VALUES FROM (10001) TO (15000);
```

Lesson 10 - DBLink & Foreign Data Wrappers

DBLINK

Dblink feature allows the session to connect and query remote PostgreSQL servers or we can also open a new connection to same PostgreSQL server locally. This feature comes handy for achieving cross database queries in PostgreSQL.

DBLINK Extension

To use this feature we need create dblink extension in a database

Syntax:

```
CREATE EXTENSION dblink;
```

Dblink Functions

dblink_connect(string1, string2)

Where string1 is a given name to this connection and string2 is connection string to connect to remote PostgreSQL server.

Example:

```
SELECT dblink_connect('myconn', 'host=192.168.20.100 user=scott password=secret
dbname=sales');
```

dblink_exec(string1, string2)

Where string1 is the connection name and string2 is the query to be executed on remote PostgreSQL server.

Example:

```
SELECT * FROM dblink('myconn', 'SELECT id, name FROM remote_table') AS t(id int, name
text);
```

dblink_disconnect(string1)

Closes the open connection to the remote database

Example:

```
SELECT dblink_disconnect(‘myconn’) ;
```

Foreign Data Wrapper (FDW)

Like DBLink FDW also allows us to access remote PostgreSQL table in the current database. FDW requires the extension to be created before using this feature.

Unlike DBLink where we need to provide the connection string every time we need to access remote tables, FDW allows us to create a permanent Foreign Server point to a remote PostgreSQL server and a Permanent user mapping.

FDW Extension

Create extension by executing following query.

Example:

```
CREATE EXTENSION postgres_fdw;
```

Create Foreign Server

Syntax:

```
CREATE SERVER <name>  FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host ‘ip/hostname,
dbname ‘databasename’, port nnnn);
```

Example:

```
CREATE SERVER myserver  FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host ‘192.168.20.100’,
dbname ‘sales’, port '5432');
```

User Mapping

Once the Foreign Server is created we can map user to access remote tables.

Syntax:

```
CREATE USER MAPPING FOR local_user  SERVER foreign_server OPTIONS (user 'remote_user',
password 'remote_password');
```

Example:

```
CREATE USER MAPPING FOR scott  SERVER myserver OPTIONS (user ‘john’, password ‘secret’);
```

Foreign Table

A table needs to be created locally which maps to a table in remote (foreign) server.

Example:

```
CREATE FOREIGN TABLE foreign_table (id integer, name text  ) SERVER myserver OPTIONS
(schema_name 'public', table_name 'remote_table');
```

Cross Database Queries

In PostgreSQL a user session can access the objects from the database to which it is connected and it cannot access object from other database. To over come this limitation we can make to use of the DBLink and FDW features of PostgreSQL.

Here what we do is, instead of creating DBLink or Foreign Server to a remote PostgreSQL server we point them to local server and access objects from other databases.

Lesson 11 - Server Side Programming in PostgreSQL

Server side programming is possible using PL/pgSQL language which comes by default as a pre installed extension in PostgreSQL. It allows to implement business rules at server side rather in application.

Originally PostgreSQL supported only Triggers & user defined Functions but support for stored procedures is added in PostgreSQL version 11. In older versions Functions used to serve as both Functions which can return a value and also as stored procedure which does not return any value but executes the given code block.

One of the advantage of stored procedures is that it allows autonomous transaction within a stored procedure.

General Structure of a program block

```
create [or replace] procedure/function  <name>(param_list)
   [returns return_type]
   language plpgsql
  as
$$
declare
-- variable declaration
begin
 -- logic
exception
 -- exception handling
end;
$$
```

Stored Procedures

General Syntax of Stored procedure:

```
create [or replace] procedure procedure_name(parameter_list)
language plpgsql
as $$
declare
-- variable declaration
begin
-- stored procedure body
[exception]
-- exception handling
    RETURN;
end; $$
```

Syntax Explanation:

First, specify the name of the stored procedure after the `create procedure` keywords.

Second, define parameters for the stored procedure. A stored procedure can accept zero or more parameters.

Third, specify `plpgsql` as the procedural language for the stored procedure. Note that you can use other procedural languages for the stored procedure such as SQL, C, etc.

Fourth, use the dollar-quoted string constant syntax to define the body of the stored procedure.

Parameters in stored procedures can have the in and inout modes parameters. They cannot have the out mode as currently this mode is not supported.

Fifth, A stored procedure does not return a value. You cannot use the return statement with a value inside a store procedure but can be used without values.

A stored procedure can have IN, INOUT parameters.

Executing a Stored Procedure

```
call stored_procedure_name(argument_list);
```

Functions

Basically in PostgreSQL Functions and Procedures are same until version 11. We can define a function which can return a value or also we can define function which does not return any values in PostgreSQL.

Syntax:

```
create [or replace] function function_name(param_list)
   returns return_type
   language plpgsql
  as
$$
declare
-- variable declaration

begin
 -- logic
 RETURN <Value>;
exception
 -- Exception Handling
    [RETURN <Value>];
end;
$$
```

Syntax Explanation

First, specify the name of the function after the create function keywords. If you want to replace the existing function, you can use the or replace keywords.

Then, specify the function parameter list surrounded by parentheses after the function name. A function can have zero or many parameters.

Next, specify the datatype of the returned value after the returns keyword.

After that, use the language plpgsql to specify the procedural language of the function. Note that PostgreSQL supports many procedural languages, not just plpgsql.

Finally, place a block in the dollar-quoted string constant.

Parameters in functions can have the IN, OUT and INOUT modes parameters.

Function Overloading

PostgreSQL allows to define more than one function with the same name as long as the arguments they take are different. This is called as function overloading.

The server will determine which function to call from the data types and the number of the provided arguments.

Writing PL/pgSQL Code

Variable Declaration

Declare block is used to declare all required variables in PostgreSQL programming. Undeclared variables causes error and terminates the execution of program.

Syntax:

```
varname [ CONSTANT ] type  [ NOT NULL ] [ { DEFAULT | := | = } expression ];
```

All variables declared as NOT NULL must have a notnull default value specified.

Equal (=) can be used instead of PL/SQL compliant :=.

The CONSTANT option prevents the variable from being assigned to after initialization

If the DEFAULT clause is not given then the variable is initialized to the SQL null value.

Examples:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

Variable Aliases

PL/pgSQL allows procedures or functions to have arguments (parameters) without names just by mentioning datatypes.

For Example:

```
CREATE PROCEDURE  test (int, text) AS
```

In such case the parameters can be referred within code block using identifiers `$1, $2...` etc.

For convenience and readability PL/pgSQL allows to create aliases for such identifiers

Syntax:

```
varname  ALIAS FOR  $n
```

Example:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
```

```
$$ LANGUAGE plpgsql;
```

Variable Aliasing

ALIAS can also be used to assign an additional name to a variables

```
DECLARE
  X int;
  Y ALIAS for X;
```

Variable Assignment

`:=` is the assignment operator in PL/pgSQL

Syntax:

```
varname  :=  expression | Constant
Examples
X :=  10;
Y :=  X + 1;
```

Copying Types

`%TYPE` allows to declare a variable with same data type as a columns datatype from a table

Syntax

```
varname   <tablename>.<columnsname>%TYPE;
```

Example:

```
salary  emp.sal%TYPE;
```

`%TYPE` can also be used to declare a variable's datatype which matches other variable

Example

```
Declare
  X  int;
  Y  X%TYPE;
```

Rowtype Variables

A rowtype variable can be declared to have the same type as the columns of an existing table or view, by using the table_name `%ROWTYPE` notation

Syntax:

```
varname   table_name%ROWTYPE;
```

Example:

```
emp_rec  emp%ROWTYPE;
```

Record Types

Record variables are similar to row-type variables, but they have no predefined structure. They take on the actual row structure of the row they are assigned during a SELECT or FOR command.

Syntax:

```
DECLARE
  varname  RECORD;
```

Example

```
DECLARE
   emp_rec RECORD;
BEGIN
   select * into emp_rec from emp limit 1;
 ....
...
```

Arithmetic Operations

All arithmetic operations can be performed by using standard arithmetic operators and assignment operator

Example:

```
DECLARE
   x int;
   y int;
   z int;
BEGIN
  z  :=  x + y;
```

Operators:

```
+Addition, - Subtract, * Multiply, / Divide, % Modulo, ^ Exponent, |/ Square Root, ||/
Cube root
```

Conditional Statements

Conditional Statements allows conditional execution of code based on certain boolean condition.

IF Conditions

IF conditions determines the block of statements to be executed or not depending on a boolean condition.

Syntax:

```
IF ... THEN ... END IF;

IF ... THEN ... ELSE ... END IF;

IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF;
```

Example:

```
BEGIN
  SELECT  count(*) INTO cn  FROM emp;
  IF  cn > 10 THEN
      SELECT "More than 10 Employees";
  ELSE
      SELECT "Less than 10 Employees";
  END IF;
END;
```

CASE Construct

CASE constructs are useful when we are examining the possibilities of multiple options.

PL/pgSQL provides 2 variation in CASE statement called as Simple CASE and Searched CASE.

Simple CASE

A simple case allows us to examine the values of a variable / expression based in equality operand. If none of the match found then ELSE block is executed.

Syntax:

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Example:

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    WHEN 3  THEN
        msg := 'Its 3';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

Searched CASE

The searched CASE provides conditional execution based on truth of given Boolean expressions. If none of the expressions are true then ELSE block is executed.

Syntax:

```
CASE
    WHEN boolean-expression THEN
```

```
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

Example:

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

Note: In both forms of CASE omitting ELSE block may result in exception CASE_NOT_FOUND if none of the given matches are found. So it is necessary to provide ELSE block

LOOPs in PL/pgSQL

Loops allows execution of a code block several times repetitively.

PL/pgSQL provides 4 types of loops they are

- Simple LOOP
- WHILE LOOP
- FOR LOOP
- FOREACH LOOP

Simple LOOP

The LOOP defines an unconditional loop that executes a block of code repeatedly until terminated by an EXIT or RETURN statement. Basically it is an infinite loop.

Syntax:

```
LOOP
    statements
END LOOP;
```

To terminate the loop **EXIT** statement must be used.

WHILE LOOP

WHILE LOOP repeats the code block as long as the given boolean expression remains true and exits when the given boolean expression turns false.

Syntax:

```
WHILE boolean-expression LOOP
```

```
    statements
END LOOP;
```

Example:

```
WHILE  x < 10 LOOP
  raise notice 'x = %', x;
END LOOP;
```

FOR LOOP

For loop can be used to iterate a block of code over a range of values

Syntax:

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;


FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;


FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

Lower bond must be less than higher bond other wise loop is not executed and exception is raised.

In case if the lower bond is grater than higher bond then REVERSE must be used as shown in syntax.

Example:

```
create or replace procedure myproc() language plpgsql as
$$
begin
 for i in 1..10 loop
raise notice 'i is %', i;
 end loop;
end;
 $$
```

Execution

```
call myproc();
```

FOR LOOP with QUERY

If a query is used instead of range of values then the block of code is executed for each for returned by the query. It behaves like a cursor.

Syntax:

```
FOR target IN query LOOP
    statements
END LOOP;
```

Example:

```
create or replace procedure myproc2() language plpgsql as
$$
declare i record;
begin
 for i in select * from emp loop
raise notice 'i is %', i.ename;
 end loop;
end;
 $$ ;
```

Execution

```
call myproc2();
```

FOREACH LOOP

FOREACH loop is designed for iterating through elements of an arrays. It behaves like for loop.

Syntax:

```
FOREACH target  IN ARRAY expression LOOP
    statements
END LOOP;
```

Example:

```
CREATE FUNCTION sum(int[]) RETURNS int AS $$
DECLARE
  s int8 := 0;
  x int;
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

Execution

```
select sum(array [1,2,3]);
```

Cursors

PL/pgSQL provides cursor for fetching table data into cursor and then allowing it to read row by row within a code block. Cursor behaves much similar like FOR loop with query.

Using a cursor involves

- Declaring a Cursor
- Opening Cursor
- Fetching data from cursor
- Closing Cursor

Declaring a Cursor

Cursor definition needs to be declared in DECLARE block before using the cursor. It usually involves a cursor variable whose datatype will be refcursor and the query.

Syntax:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR / IS query;
```

`SCROLL` cursors allows backward fetches.

Arguments allows you to specify variables to be used when opening the cursor

Example:

```
DECLARE
      c1   CURSOR FOR select * from emp;
      c2   CURSOR   (dn   int) FOR select * from emp where deptno=dn;
      c3   CURSOR refcursor;
```

Last one `(c3)` is declared without query. A query can also be specified at the time of opening the cursor and such cursors are called as unbound cursors.

Opening Cursor

Only a previously declared cursor can be opened.

Syntax:

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

Example:

```
begin
OPEN c3 FOR select * from emp;
OPEN c1;
OPEN c2(20);
```

FETCHING rows from cursor

FETCH command can be used to fetch rows from cursor and store data into variables.

Syntax:

```
FETCH [ direction { FROM | IN } ] cursor INTO target variables;
```

Examples:

```
FETCH  c1 INTO  recvar (recvar  should be record)
FETCH  c2 INTO var1,var2,var3,var4,var5,var6,var7,var8;
FETCH RELATIVE -2 FROM c3 INTO recvar;
FETCH LAST from c3 INTO recvar;
```

MOVE Cursor Position

MOVE repositions the cursor without fetching any data.

Syntax:

```
MOVE [ direction { FROM | IN } ] cursor;
```

Example:

```
MOVE  c1;
MOVE RELATIVE -2 FROM c2;
MOVE FORWARD 3 FROM c2;
MOVE LAST FROM c3;
```

UPDATE / DELETE table

Updating or Deleting the row in table where the current cursor position is.

Syntax:

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

Example:

```
UPDATE emp SET sal = sal + 1000 WHERE CURRENT OF c1;
```

Closing the CURSOR

Closing cursor frees up the resources used by CURSOR and also once a cursor is closed it can be opened again.

Syntax:

```
CLOSE  cursor;
```

Example:

```
CLOSE c1;
```

CURSOR as RETURN value

A Function can return cursor to calling block as variable of `refcursor` datatype.

Transaction Management

Transactions can be done within procedures and can be ended with **COMMIT** or **ROLLBACK** statements. Multiple transactions are possible within same stored procedure.

Within a stored procedure there is no need to start transaction explicitly with **START TRANSACTION** as new transaction is started whenever a procedure is called using **CALL** statement.

Within a stored procedure if a transaction is completed then a new transaction is automatically started implicitly.

- Opened cursors are closed when the transaction is completed.
- A transaction cannot be ended inside a block with exception handlers.

Exception Handling in PL/pgSQL

Raise statement can be used for exception handling or error reporting and also for printing messages.

Syntax:

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

Where level is one of the following

- debug
- log
- notice
- info
- warning
- exception

The default level is "exception" and it is also the level which raises the error resulting in program termination.

Explicitly raised exceptions can be handled within the **BEGIN** block or in **EXCEPTION** block whereas implicit exception can be handled in **EXCEPTION** block.

Implicit PL/pgSQL Exception

```
P0000  plpgsql_error
P0001  raise_exception
P0002  no_data_found
P0003  too_many_rows
```

no_data_found and **too_many_rows** are raised implicitly only if the **STRICT** clause is used with **SELECT** as this condition is only a warning in SQL.

Example:

```
declare
   r record;
```

```
    .....
    .....
begin
   .....
   .....
   select * into strict r from emp where empno = en;
   .....
exception
   when no_data_found then raise exception 'Invalid empno %', en;
   when too_many_rows then raise exception 'More than one employee
has empno %', en;
end;
```

Explicitly Raised Exceptions

Exceptions can be raised explicitly within BEGIN block. For example look at the code below

```
BEGIN
    ....
    if ( qty < 10 )  then
       raise exception 'minimum accepted quantity is 10, Given is %', qty;
    end if;
END;
```

Or it can also be done in the below mentioned way

```
BEGIN
    ....
    if ( qty < 10 )  then
       raise exception  using errcode = 22000;
    end if;
exception
    on data_exception the raise exception 'minimum accepted quantity is 10, Given is %',
qty;
END;
```

Note: in above example the ERROR CODE for condition data_exception is 22000. See all error codes and corresponding condition names in given ERROR CODE table.

Triggers

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed.

Supported Trigger Events

- INSERT
- UPDATE
- DELETE
- TRUNCATE

Timing

- BEFORE
- AFTER

Trigger Types

- FOR EACH STATEMENT
- FOR EACH ROW

Syntax

```
CREATE TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event}
    ON table    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )
```

where event can be one of:

- INSERT
- UPDATE [OF column_name [, ...]]
- DELETE
- TRUNCATE

Update triggers can have a list of columns specified so that the triggers is fired only when these columns are changed.

Trigger Function

In postgresql when a trigger is fired it executes a function and the function contain the code block needed by the trigger. Tigger Function must be created before creating a trigger.

This function must return a trigger type and must not have any arguments.

Trigger Function Parameters

Whenever trigger executes the trigger function it passes following parameters implicitly.

NEW

Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is null in statement-level triggers and for DELETE operations.

OLD

Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is null in statement-level triggers and for INSERT operations.

TG_NAME

Variable that contains the name of the trigger actually fired.

TG_WHEN

A string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.

TG_LEVEL

A string of either ROW or STATEMENT depending on the trigger's definition.

TG_OP

A string of INSERT, UPDATE, DELETE, or TRUNCATE telling for which operation the trigger was fired.

TG_RELID

The object ID of the table that caused the trigger invocation.

TG_RELNAME

Data type name; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use TG_TABLE_NAME instead.

TG_TABLE_NAME

The name of the table that caused the trigger invocation.

TG_TABLE_SCHEMA

The name of the schema of the table that caused the trigger invocation.

Only the record variable `NEW` is modifiable in the trigger function.

Function Return Value.

Function can return a record type which matches row operation or can also return NULL in which case the current row operation is not performed.

A statement level trigger function must return NULL as there is no operation being done on individual rows.

FOR EACH ROW trigger can return NEW or OLD and any other Record type variable.

Instead Of Triggers

INSTEAD OF triggers are created only on views. These triggers must be FOR EACH ROW as statement level triggers are not supported on views.

Trigger Example

Lets say that a trigger is needed on emp table to make sure that the columns ename & job always have values in upper case letters.

Trigger Function

```
create or replace function emp_trig1()  returns trigger
LANGUAGE plpgsql as
$$
begin
   new.ename :=  upper(new.ename);
   new.job := upper(new.job);
   return new;
end;  $$ ;
```

Trigger statement for update

```
create  trigger emptrig1  before update of ename, job on emp for each row  execute
procedure emp_trig1();
```

Trigger statement for Insert

```
create  trigger emptrig2  before insert on emp for each row  execute procedure
emp_trig1();
```

Event Triggers

Unlike the triggers which are associated with a particular table event triggers are global objects associated with databases and are capable of capturing DDL events.

Event triggers are fired for the following events

- ddl_command_start
- ddl_command_end
- sql_drop

ddl_command_start

The ddl_command_start event occurs just before the execution of a CREATE, ALTER, or DROP command.

This event is not applicable for share object like database, tablespace, user/role etc

ddl_command_end

The ddl_command_end event occurs just after the execution of this same set of commands

sql_drop

The sql_drop event occurs just before the ddl_command_end event trigger for any operation that drops database objects.

Syntax

```
CREATE EVENT TRIGGER name  ON event  EXECUTE PROCEDURE function_name()
```

Autonomous Transactions in PostgreSQL

Autonomous transactions are like a transaction from within a transaction subprogram to execute SQL operations and commit or rollback these operations without committing or rolling-back the main transaction.

PostgreSQL does not explicitly support Autonomous Transactions and it is recommend to refactoring is done to eliminate the requirement of using Autonomous Transactions.

In case, eliminating is not an option then the following workaround will be helpful to achieve the pragma autonomous transaction effect using dblink feature of PostgreSQL.

DBLINK

Dblink feature allows the session to connect and query remote PostgreSQL servers or we can also open a new connection to same PostgreSQL server locally. This feature comes handy for achieving Autonomous Transactions in PostgreSQL.

Where string1 is the connection name.

Example of Autonomous transaction

Write a function for Autonomous transaction (Inner Transaction) where we perform transaction using dblink.

Lets say from this function I want to INSERT a row in DEPT table using dblink.

Function Code

```
CREATE OR REPLACE FUNCTION myfun1()
RETURNS void
AS
$BODY$
DECLARE
    v_sql text;
BEGIN
    PERFORM dblink_connect('myconn',
         'dbname=employees user=postgres password=postgres host=127.0.0.1 port=5432');
    v_sql := format('insert into dept values (%s, %L, %L)', 60,'HR','DENVER');
    PERFORM dblink_exec('myconn', v_sql);
    PERFORM dblink_disconnect('myconn');
END;
$BODY$
LANGUAGE  plpgsql;
```

Now that the Function is created we can go ahead and a transaction and from within that transaction we call the above function.

Here note that the function myfun1() independently commits the transaction where as the outer transaction is rollbacked

Autonomous Transaction

```
select * from dept;
begin;
insert into dept values (70,'CLOUD','SCOTTDALE');
select myfun1();
rollback;
select * from dept;
```

This works fine even if the database or tables in both transactions differs.

Lesson 12 – Database Security

Database Users / Roles

- Roles are useful in managing database access permissions.
- A role can be considered as Database User or a Database Group
- Roles can own objects and controls who has access to which object.
- It is possible to grant membership in a Role to another Role .
- Roles in postgres are mixture of both USER & GROUP concept
- Any Role can act as a USER, GROUP or both
- Database Roles are global across cluster

Creating a Role / User

```
SQL: CREATE ROLE  <rolename>;
```

- To create a role using utility program

```
SQL: CREATE USER  <rolename>;
```

Create a User or ROLE using utility program

```
CMD: createuser  <rolename>;
```

ROLE Attributes

Role attributes define the User/Roles database cluster wide permissions. Attributes can only be added or removed to Roles/User using ALTER USER command or while can be added while creating User/Role.

User/Role Attributes

```
| SUPERUSER      | NOSUPERUSER
| CREATEDB       | NOCREATEDB
| CREATEROLE     | NOCREATEROLE
| INHERIT        | NOINHERIT
| LOGIN          | NOLOGIN
| REPLICATION    | NOREPLICATION
| BYPASSRLS      | NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| ROLE  role_name [, ...]
```

Renaming a USER or ROLE

```
SQL: ALTER ROLE  <rolename>  RENAME TO <NewNAME>;
```

Locking Account

Locking a user account is nothing but assign NOLOGIN attribute to the user.

```
ALTER USER <user name> WITH NOLOGIN;
```

Expiring Passwords

Passwords can be expired by using the VALID UNTIL attribute.

```
ALTER USER <user name> WITH VALID UNTIL 'date';
```

USER vs ROLE

- Role is a database user or it can also be database group or both
- A user is a role with LOGIN attribute.
- CREATE USER is nothing but an alias to CREATE ROLE.
- When a role is created by CREATE USER it will have LOGIN Attribute by default
- When a role is created by CREATE ROLE it will not have LOGIN attribute by default however it can be assigned later

```
ALTER ROLE <rolename>  LOGIN;
```

Superuser

- Role attribute superuser determines if a Role is Super user or Not.
- A superuser can by pass all security checks except right to Login.

Creating a Superuser.

Syntax:

```
create role admin_user with superuser login password 'secret';
```

Role Membership

- For Better management of privileges it is advised to use groups/roles
- A role can be granted all required privileges and then it can be granted to another role.
- Ideally role which is used as group should not have LOGIN attribute
- Role with INHERIT attribute will have all privileges directly assigned to it plus the privileges it is member of.
- When a role is granted to another role then the later will become member of first role.

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

Example – Role Memebership

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

- In this example, role joe, admin and wheel are created
- Role joe is the member of admin role (group)
- Role admin is the member of wheel role (group)
- Indirectly role joe is also member of wheel role (group)
- Role joe inherits privileges from admin group
- But role admin does not inherit any privileges from wheel as NOINHERIT
- When role joe connects it will have all the privileges granted directly to it plus it will also have privileges granted to admin role as role joe has INHERIT attribute
- Any privileges granted to wheel are not available because the admin role has NOINHERIT attribute

Example – Changing Role in Session

- When a role connects and granted session it can switch role which it is member of

SET ROLE can be used to switch the roles during session

```
SET ROLE <rolename>;
```

Example:

```
psql  -U joe -W postgres
```

- Now the session has all privileges granted to joe and admin

```
Postgres=> set role admin;
```

- Now the session has all privileges granted to admin role only because admin has NOINHERIT

attribute

```
Postgres=> set role wheel;
```

- Now the session has all privileges granted to wheel role only because wheel has NOINHERIT

attribute

- When role joe connects it will have all the privileges granted directly to it plus it will also have privileges granted to admin role as role joe has INHERIT attribute
- Any privileges granted to wheel are not available because the admin role has NOINHERIT attribute

Example – Resetting Role in Session

- We reset the role in session to go back original connection role

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

NOTE: The role attributes LOGIN, SUPERUSER, CREATEDB, and CREATEROLE are special privileges which cannot be inherited

Built-in Roles

Built-in roles are introduced from version 15 onwards. These roles can be granted to any user.

Predefined Roles

• Role	• Allowed Access
• pg_read_all_data	• Read all data (tables, views, sequences), as if having SELECT rights on those objects, and USAGE rights on all schemas, even without having it explicitly. This role does not have the role attribute BYPASSRLS set. If RLS is being used, an administrator may wish to set BYPASSRLS on roles which this role is GRANTed to.
• pg_write_all_data	• Write all data (tables, views, sequences), as if having INSERT, UPDATE, and DELETE rights on those objects, and USAGE rights on all schemas, even without having it explicitly. This role does not have the role attribute BYPASSRLS set. If RLS is being used, an administrator may wish to set BYPASSRLS on roles which this role is GRANTed to.
• pg_read_all_settings	• Read all configuration variables, even those normally visible only to superusers.
• pg_read_all_stats	• Read all pg_stat_* views and use various statistics related extensions, even those normally visible only to superusers.
• pg_stat_scan_tables	• Execute monitoring functions that may take ACCESS SHARE locks on tables, potentially for a long time.
• pg_monitor	• Read/execute various monitoring views and functions. This role is a member of pg_read_all_settings, pg_read_all_stats and pg_stat_scan_table s.
• pg_database_owner	• None. Membership consists, implicitly, of the current database owner.
• pg_signal_backend	• Signal another backend to cancel a query or terminate its session.
• pg_read_server_files	• Allow reading files from any location the database can access on the server with COPY and other file-access functions.

• Role	• Allowed Access
• pg_write_server_files	• Allow writing to files in any location the database can access on the server with COPY and other file-access functions.
• pg_execute_server_program	• Allow executing programs on the database server as the user the database runs as with COPY and other functions which allow executing a server-side program.
• pg_checkpoint	• Allow executing the CHECKPOINT command.
• pg_maintain	• Allow executing VACUUM, ANALYZE, CLUSTER, REFRESH MATERIALIZED VIEW, REINDEX, and LOCK TABLE on all relations, as if having MAINTAIN rights on those objects, even without having it explicitly.
• pg_use_reserved_connection s	• Allow use of connection slots reserved via **reserved_connections**.
• pg_create_subscription	• Allow users with CREATE permission on the database to issue CREATE SUBSCRIPTION.

Example:
Lets allow user named “maint” to perform maintainance activities such as vacuuming etc.

```
GRANT pg_maintain TO maint;
```

Dropping Role

- Roles owns database objects which makes it complicated to drop them straight forward.
- All objects owned by role must be reassinged or dropped before dropping role

Reassigning ownership

```
REASSIGN OWNED BY doomed_role TO successor_role;
```

Dropping objects owned by role

```
DROP OWNED BY doomed_role;
```

Dropping Role

REASSIGN OWNED BY doomed_role TO successor_role;
```
DROP OWNED BY doomed_role;
```
-- repeat the above commands in each database of the cluster
```
DROP ROLE doomed_role;
```

PUBLIC Role

PostgreSQL grants privileges on some types of objects to PUBLIC by default when the objects are created.

- No privileges are granted to PUBLIC by default on tables, table columns, sequences, foreign data wrappers, foreign servers, large objects, schemas, or tablespaces.
- For Maximum security revoke the privileges granted to public immediately after creating objects like databases, .
- Every database is granted connect, temporary privilege to public.
- Every function / procedure is granted execute to public.
- Domain, type, language is granted usage to public.

Object Privileges & ACL's

PSQL command lists the privileges in form of single letter

ACLs for Objects

Privilege	**Abbreviation**	**Applicable Object Types**
SELECT	r (“read”)	LARGE OBJECT, SEQUENCE, TABLE (and table-like objects), table column
INSERT	a (“append”)	TABLE, table column
UPDATE	w (“write”)	LARGE OBJECT, SEQUENCE, TABLE, table column
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, table column
TRIGGER	t	TABLE
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGNSERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE

privileges available for each Object

PSQL command lists the privileges in form of single letter .

Object Type	All Privileges	Default PUBLIC Privileges	psql Command
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION or PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	none	\dew+
FOREIGN SERVER	U	none	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	none	
SCHEMA	UC	none	\dn+
SEQUENCE	rwU	none	\dp
TABLE (and table-like objects)	arwdDxt	none	\dp
Table column	arwx	none	\dp
TABLESPACE	C	none	\db+
TYPE	U	U	\dT+

Object Privileges

GRANT statement

- GRANTcan be used for granting object level privileges to database users, groups or roles.
- Privileges can be granted on a tablespace, database, schema, table, sequence, domain and function.
- GRANT ..... [WITH GRANT OPTION] can be used to grant only the granting option to the user.

Syntax:

```
GRANT  Privilege list  ON  <OBJECT>   TO  <USER> [ WITH GRANT OPTION ];
GRANT Privilege list  ON ALL TABLES IN SCHEMA "schema name" TO <USER> [ WITH GRANT OPTION
];
GRANT Privilege list ON DATABASE database_name TO  <USER> [ WITH GRANT OPTION ];
GRANT Privilege list ON SCHEMA schema_name TO <USER> [ WITH GRANT OPTION ];
```

ALL PRIVILEGES

- ALL key word can be used to grant all of the privileges available for the object's type.

PRIVILEGE LIST

- SELECT
- INSERT
- UPDATE
- DELETE
- TRUNCATE
- REFERENCES
- TRIGGER
- CREATE
- CONNECT
- TEMPORARY
- EXECUTE
- USAGE

REVOKE Statement

- REVOKEcan be used for revoking object level privileges to database users, groups or roles.
- Privileges can be revoked on a tablespace, database, schema, table, sequence, domain and function.
- REVOKE [GRANT OPTION FOR] can be used to revoke only the grant option without revoking the actual privilege.

Syntax:

```
REVOKE Privilege list   ON  <OBJECT>    FROM  <USER>;
```

Data dictionary Objects

- **`pg_authid`** -- All Role / User details are stored in this table
- **`pg_roles`** – details of all created role.
- **`pg_shadow`** – password details.
- **`role_table_grants`** – privileges assigned on tables .
- **`role_column_grants`** – privileges assigned on columns.
- **`role_usage_grants`** – usage privilege granted to various objects.

`routine_privileges` – privileges granted to functions/procedures

Lesson 13 : Monitoring PostgreSQL

Viewing Statistics

- PostgreSQL provides many catalog views to examine the statistics .
- These views can be queried to examine the server activity.
- When server is shutdown the statistics are persisted in pg_stat directory.
- It is advised to configure pg_stat_tmp directory to SSD storage or RAM DISK using stat_temp_directory parameter.

Viewing Dynamic Statistics

pg_stat_activity

information related to the current activity of that process, such as state and current query.

pg_stat_ssl

One row per each connection using ssl.

pg_stat_gssapi

One row per each connection using gssapi authentication.

pg_stat_progress_create_index

one row for each backend running create index or reindex.

pg_stat_progress_vacuum

one row for each backend running vacuum or autovacuum

pg_stat_progress_cluster

one for each backend running CLUSTER or VACUUM FULL.

Viewing Collected Statistics

pg_stat_database

one row per database showing database wide statistics.

pg_stat_user_tables

One row per each user table.

pg_stat_xact_user_tables

same as pg_stat_user_table but has details of current transactions.

pg_stat_user_indexes

one row for each user created index.

pg_statio_user_tables

one row for user created table shows IO related statistics.

pg_statio_user_sequences

one row for user created sequences shows IO related statistics.

pg_stat_user_functions

one row per database showing functions related statistics.

pg_stat_xact_user_functions

same as previous one but show statistics from current transactions.

pg_stat_archiver

Show stattistics about wal archiving.

pg_stat_bgwriter

Show statistics about BGWriter process

pg_stat_wal

shows stat about WAL activity

Log configuration

Following important parameters can be configured to log more details on database operations.

log_min_messages (enum)

Controls which message levels are written to the server log. Valid values are **DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, and PANIC.**

log_min_error_statement (enum)

Controls which SQL statements that cause an error condition are recorded in the server log. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, and PANIC. The default is ERROR.

log_min_duration_statement (integer)

Causes the duration of each completed statement to be logged if the statement ran for at least the specified amount of time.

log_autovacuum_min_duration (integer)

Causes each action executed by autovacuum to be logged if it ran for at least the specified amount of time. Setting this to zero logs all autovacuum actions. -1 disables.

log_checkpoints (boolean)

Causes checkpoints and restart points to be logged in the server log. The default is on.

log_connections (boolean)

Causes each attempted connection to the server to be logged. The default is off.

log_disconnections (boolean)

Causes session terminations to be logged. The log output provides information similar to log_connections, plus the duration of the session. The default is off.

log_duration (boolean)

Causes the duration of every completed statement to be logged. The default is off.

log_lock_waits (boolean)

Controls whether a log message is produced when a session waits longer than deadlock_timeout to acquire a lock.

log_statement (enum)

Controls which SQL statements are logged. Valid values are none (off), ddl, mod, and all (all statements).

log_temp_files (integer)

Controls logging of temporary file names and sizes. The default setting is -1, which disables such logging.

Database Lock Monitoring

One of the frequent causes of performance issues is database locks. A DBA must monitor locks to identify queries causing them and optimizing those queries.

Tables and functions useful to monitoring and troubleshooting database locks:

pg_stat_activity view:

A view with one entry per server process (user session), showing details of the running query for each.

pg_locks view:

Information on current locks held within the database by open transactions, with one row per lockable object.

pg_blocking_pids():

A function that can find the process IDs (PIDs) of sessions that are blocking the PostgreSQL server process of a supplied PID.

pg_cancel_backend():

Function that cancels the currently running query by sending a SIGINT to a process ID.

pg_terminate_backend():

Terminate a backend process completely (the query and usually the connection) on the database (uses SIGTERM instead of SIGINT).

Details of pg_stat_activity

This view provides the details of current session activity on the server. From this view we can track all the current session and current transactions.

Column Type Description

datid oid OID of the database this backend is connected to
datname name Name of the database this backend is connected to
pid integer Process ID of this backend
leader_pid integer Process ID of the parallel group leader, if this process is a parallel query worker. NULL if this process is a parallel group leader or does not participate in parallel query.
usesysid oid OID of the user logged into this backend
usename name Name of the user logged into this backend
application_name text Name of the application that is connected to this backend
client_addr inet IP address of the client connected to this backend. If this field is null, it indicates either that the client is connected via a Unix socket on the server machine or that this is an internal process such as autovacuum.
client_hostname text Host name of the connected client, as reported by a reverse DNS lookup of client_addr. This field will only be non-null for IP connections, and only when **log_hostname** is enabled.
client_port integer TCP port number that the client is using for communication with this backend, or -1 if a Unix socket is used. If this field is null, it indicates that this is an internal server process.
backend_start timestamp with time zone Time when this process was started. For client backends, this is the time the client connected to the server.

xact_start timestamp with time zone Time when this process' current transaction was started, or null if no transaction is active. If the current query is the first of its transaction, this column is equal to the query_start column.
query_start timestamp with time zone Time when the currently active query was started, or if state is not active, when the last query was started
state_change timestamp with time zone Time when the state was last changed
wait_event_type text The type of event for which the backend is waiting, if any; otherwise NULL.
wait_event text Wait event name if backend is currently waiting, otherwise NULL.
state text Current overall state of this backend. Possible values are: • active: The backend is executing a query. • idle: The backend is waiting for a new client command. • idle in transaction: The backend is in a transaction, but is not currently executing a query. • idle in transaction (aborted): This state is similar to idle in transaction, except one of the statements • in the transaction caused an error. • fastpath function call: The backend is executing a fast-path function. • disabled: This state is reported if **track_activities** is disabled in this backend.
backend_xid xid Top-level transaction identifier of this backend, if any.
backend_xmin xid The current backend's xmin horizon.
query text Text of this backend's most recent query. If state is active this field shows the currently executing query. In all other states, it shows the last query that was executed. By default the query text is truncated at 1024 bytes; this value can be changed via the parameter **track_activity_query_size**.

backend_type text Type of current backend. Possible types are autovacuum launcher, autovacuum worker, logical replication launcher, logical replication worker, parallel worker, background writer, client backend, checkpointer, startup, walreceiver, walsender and walwriter. In addition, background workers registered by extensions may have additional types.

Details of pg_lock view

Provides information about the currently granted locks and currently waiting locks.

Column Type Description
locktype text Type of the lockable object: relation, extend, frozenid, page, tuple, transactionid, virtualxid, spectoken, object, userlock, or advisory.
database oid (references pg_database.oid) OID of the database in which the lock target exists, or zero if the target is a shared object, or null if the target is a transaction ID
relation oid (references pg_class.oid) OID of the relation targeted by the lock, or null if the target is not a relation or part of a relation
page int4 Page number targeted by the lock within the relation, or null if the target is not a relation page or tuple
tuple int2 Tuple number targeted by the lock within the page, or null if the target is not a tuple
virtualxid text Virtual ID of the transaction targeted by the lock, or null if the target is not a virtual transaction ID
transactionid xid ID of the transaction targeted by the lock, or null if the target is not a transaction ID

classid oid (references pg_class.oid) OID of the system catalog containing the lock target, or null if the target is not a general database object
objid oid (references any OID column) OID of the lock target within its system catalog, or null if the target is not a general database object
objsubid int2 Column number targeted by the lock (the classid and objid refer to the table itself), or zero if the target is some other general database object, or null if the target is not a general database object
virtualtransaction text Virtual ID of the transaction that is holding or awaiting this lock
pid int4 Process ID of the server process holding or awaiting this lock, or null if the lock is held by a prepared transaction
mode text Name of the lock mode held or desired by this process
granted bool True if lock is held, false if lock is awaited
fastpath bool True if lock was taken via fast path, false if taken via main lock table

Useful Queries to monitor and troubleshoot locking issues

Find queries running more than 2 minutes.

```
SELECT pid, user, query_start, (now() - query_start) AS query_time, query, state,
wait_event_type, wait_event FROM pg_stat_activity WHERE (now() - query_start) > interval
'2 minutes';
```

Blocking and blocked queries

Following query list out the blocking query, blocked query, duration.

```
SELECT COALESCE(blockingl.relation::regclass::text,blockingl.locktype) as locked_item,
now() - blockeda.query_start AS waiting_duration, blockeda.pid AS blocked_pid,
blockeda.query as blocked_query, blockedl.mode as blocked_mode, blockinga.pid AS
blocking_pid, blockinga.query as blocking_query, blockingl.mode as blocking_mode FROM
pg_catalog.pg_locks blockedl JOIN pg_stat_activity blockeda ON blockedl.pid =
blockeda.pid JOIN pg_catalog.pg_locks blockingl  ON((
(blockingl.transactionid=blockedl.transactionid) OR (blockingl.relation=blockedl.relation
AND blockingl.locktype=blockedl.locktype)) AND blockedl.pid != blockingl.pid) JOIN
pg_stat_activity blockinga ON blockingl.pid = blockinga.pid AND blockinga.datid =
blockeda.datid WHERE NOT blockedl.granted AND blockinga.datname = current_database();
```

Blocked query and backend pid details

```
select pid,  usename, pg_blocking_pids(pid) as blocked_by,  query as blocked_query from
pg_stat_activity where cardinality(pg_blocking_pids(pid)) > 0;
```

Statement Digest

PostgreSQL provides an extension called `pg_stat_statements` to collect statements level statistics and save as digest for analyzing performance of individual statements.

Setup

In postgresql.conf

```
shared_preload_libraries = 'pg_stat_statements.so'
```

Note : This change needs a service restart.

Install the extension using following command.

```
mydb=> create extension pg_stat_statements;
```

Once the extension created a view called `pg_stat_statements` is created and which contain statistics related to statements. A Row per statement executed on server.

Lesson 14 - Loading & Moving data

SQL Dumps (Logical Backups)

Logical backup are used for adhoc backup request such as moving data from production to test etc. Logical backups allows us to perform partial backups and restore.

pg_dump backups

- Generate a text file with SQL commands
- Postgresql provides the utility program **pg_dump** for this purpose.
- **pg_dump** does not block readers or writers.
- **pg_dump** does not operate with special permissions.
- Dumps created by **pg_dump** are internally consistent, that is, the dump represents a snapshot of the database as of the time **pg_dump** begins running.

Syntax:

pg_dump [options] [dbname]

pg_dump Options

-a – Data only. Do not dump the data definitions (schema)

-s – Data definitions (schema) only. Do not dump the data

-n <schema> - Dump from the specified schema only

-t <table> - Dump specified table only

-f <path/file name.backup> - Send dump to specified file

-Fp – Dump in plain-text SQL script (default)

-Ft – Dump in tar format

-Fc – Dump in compressed, custom format

-v – Verbose option

-o use oids

Restore – SQL Dump

The text files created by pg_dump are intended to be read in by the psql program. The general command form to restore a dump is:

```
$ psql dbname    <      <pgdump backup file>
```

pg_restore

`pg_restore` is used to restore a database backed up with `pg_dump's -Ft` or `-Fc` options only i.e., a non-text format.

Files are portable across architectures and PostgreSQL versions.

Syntax:

```
pg_restore [options…] [pgdump backup file]
```

pg_restore Options

-d <database name>: Connect to the specified database. Also restores to this database if –C option is omitted.

-C: Create the database named in the dump file & restore directly into it.

-a: Restore the data only, not the data definitions (schema).

-s: Restore the data definitions (schema) only, not the data.

-n <schema>: Restore only objects from specified schema.

-t <table>: Restore only specified table.

-v: Verbose option.

Entire Cluster – SQL Dump

- `pg_dumpall` is used to dump an entire database cluster in plain-text SQL format
- Dumps global objects: user, groups, and associated permissions
- Use psql to restore

Syntax:

```
pg_dumpall [options…] > filename.backup
```

pg_dumpall Options

-a: Data only. Do not dump schema.

-s: Data definitions (schema) only.

-g: Dump global objects only - not databases.

-r: Dump only roles.

-c: Clean (drop) databases before recreating.

-O: Skip restoration of object ownership.

-x: Do not dump privileges (grant/revoke)

--disable-triggers: Disable triggers during data-only restore.

-v: Verbose option.

Exporting / Importing Data

COPY command can be used for both importing and exporting data from / to a table to CSV, TEST or BINARY files.

Exporting

A table data can be exported to a portable CSV or TAB Separated text format. PostgreSQL gives flexibility of choosing which columns to include or can also specify a query instead of Table.

```
Syntax 1:
COPY table_name ( column_name [, ...] ) TO  'filename' WITH  Options…..;
Syntax 2:
COPY (  SELECT query  )  TO  'filename'  WITH  Options…..;
Syntax 3:
COPY (  SELECT query  )  TO  PROGRAM 'command'  WITH  Options…..;
Syntax 4:
COPY table_name ( column_name [, ...] ) TO  PROGRAM 'command'  WITH  Options…..;

OPTIONS:
CSV
TEXT
DELIMITER <character>
Null  <null replacement>
QUOTE ,quotation character>
HEADERS {TRUE | FALSE}
```

Example of Exporting Emp table:

In this example EMP table is exported to `'/var/lib/pgsql/emp.csv'` file in CSV format

```
postgres=# copy emp to '/var/lib/pgsql/emp.csv' with CSV delimiter ',' null 'NULL' quote
'"' ;
COPY 14
```

Importing

Any CSV or TAB Separated text file can be imported into a table If the column count and data types matches irrespective of the origin of the file.

```
COPY  <TableName>(Column1, Column2, ….)  FROM  /path/to/CSVfile WITH options.. [ WHERE
condition ];
```

Options are same a Exporting options.

Example of Importing into emp_test table:

```
postgres=# copy emp from  '/var/lib/pgsql/emp.csv' with CSV delimiter ',' null 'NULL'
quote '"' ;
COPY 14
```

Table to Table Data Transfer

Copy Data from Existing Table

Insert statement allows inserting data from a table to another using SELECT query provided provided the datatype and constraints are not in conflict

Syntax:

```
INSERT INTO target_table SELECT * FROM source_table [WHERE ..];
```

Create New table and Insert Data from Existing Table

Create table statement allows to create a new table based on a SELECT Query.

Syntax:

```
create table new_table as select * from source_table ;
```

Create an Empty Table like Existing Table

We can create a new empty table with same structure as any of the existing tabl.

Syntax:

```
CREATE TABLE new_table (LIKE  existing_table [INCLUDIGN ALL]);
```

Lesson 15 - Performance Tuning

Statement Digest

PostgreSQL provides an extension called `pg_stat_statements` to collect statements level statistics and save as digest for analyzing performance of individual statements.

Setup

In postgresql.conf

```
shared_preload_libraries = 'pg_stat_statements.so'
```

Note : This change needs a service restart.

Install the extension using following command.

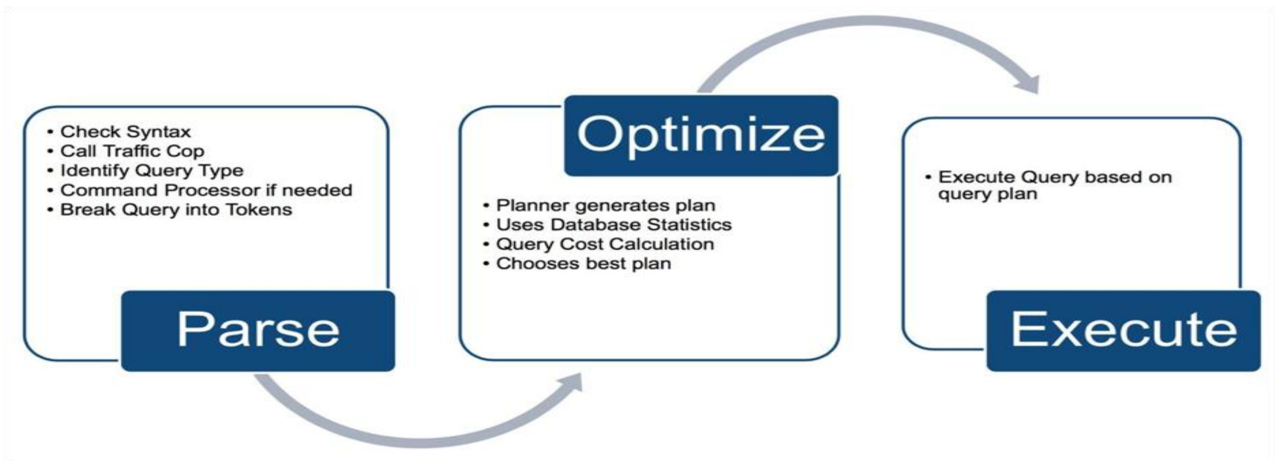
```
mydb=> create extension pg_stat_statements;
```

Once the extension created a view called `pg_stat_statements` is created and which contain statistics related to statements. A Row per statement executed on server.

Query Tuning

- Statement Processing
- Common Query Performance Issues
- SQL Tuning Goals
- SQL Tuning Steps
  1. Identify slow queries
  2. Review the query execution plan
  3. Optimizer statistics and behavior
  4. Restructure SQL statements
  5. Indexes
  6. Review Final Execution Plan

Statement Processing



Common Query Performance Issues

- Full tables scans
- Bad SQL
- Sorts using disk
- Old or missing statistics
- I/O issues
- Bad connection management

SQL Tuning Goals

- Identify bad or slow SQL
- Find the possible performance issue in a query
- Reduce total execution time
- Reduce the resource usage of a query
- Determine most efficient execution plan
- Balance or parallelize the workload

Six Step Query Tuning

- Identify Slow Queries
- Review the Query Execution Plan
- Optimizer Statistics and Behavior
- Restructure SQL Statements
- Add / Remove Indexes
- Review Final Execution Plan

STEP 1: Identify Slow Queries

`log_min_duration_statement`

This parameter sets a minimum statement execution time (in milliseconds) that causes a statement to be logged.

All SQL statements that run for the time specified or longer will be logged with their duration.

Enabling this option can be useful in tracking down non-optimized queries in your applications.

Not all queries shown will be problematic. Some large queries that are already optimized will also be logged. For example, reporting queries, large batch jobs, etc.

STEP 2: Review the Query Execution Plan

- An execution plan shows the detailed steps necessary to execute a SQL statement
- Planner is responsible for generating the execution plan

- The Optimizer determines the most efficient execution plan
- Optimization is cost-based, cost is estimated resource usage for a plan
- Cost estimates rely on accurate table statistics, gathered with ANALYZE
- Costs also rely on `seq_page_cost`, `random_page_cost`, and others
- The `EXPLAIN` command is used to view a query plan
- `EXPLAIN ANALYZE` is used to run the query to get actual runtime stats

Optimizer Statistics

- The PostgreSQL Optimizer and Planner use table statistics for generating query plans
- Choice of query plans are only as good as table stats

Table statistics

- Stored in catalog tables like pg_class, pg_stats etc.
- Stores row sampling information such as number of rows, distribution frequency, etc.
- Maintained by Autovacuum, or can be refreshed manually with ANALYZE
- Run ANALYZE after bulk data changes to ensure good plans

Query Execution plan

Query Planner creates possible plans based on the optimizer statistics and Optmimizer select the best plan for execution. The selected Query plan can be examined by using EXPLAIN statement.

Example:

```
postgres=# EXPLAIN SELECT * FROM emp;
QUERY PLAN
-------------------------------------------------------
Seq Scan on emp     (cost=0.00..1.14 rows=14 width=135)
```

The numbers that are quoted by EXPLAIN are:

Cost: 0.00 Estimated startup cost

1.14 Estimated total cost (This is the number you should aim to minimize)

Rows: 14 Estimated number of rows output by this plan node

Width: 135 Estimated average width (in bytes) of rows output by this plan node

Seq Scan on emp : tells that sequential scan method is used.

If query uses multiple tables then the above information is provided for each table separately.

Reviewing Explain Plans

- Examine the various costs at different levels in the explain plan
- Look for sequential scans on large tables
- All sequential scans are not inefficient
- Check whether a join type is appropriate for the number of rows returned

- Check for indexes used in the explain plan
- Examine the cost of sorting and aggregations
- Review whether views are used efficiently

Step 3 - Optimizer Statistics and Behavior

Table Statistics

- Absolutely critical to have accurate statistics to ensure optimal query plans
- Are not updated in real time, so should be manually updated after bulk operations
- Can be updated using `ANALYZE` command or OS command vacuumdb with -Z option
- Stored in `pg_class` and `pg_statistics`
- You can run the `ANALYZE` command from psql on specific tables and just specific columns
- Autovacuum will run `ANALYZE` as configured

Syntax:

```
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

- Planner relies on Statistics to estimate the rows retrieved by query
- Absolutely necessary to have accurate statistics
- pg_class tables relpages & reltuples provides estimates of total entries
- Information provides by relpages & reltuples is slightly outdated as they are not calculated on fly
- Most queries retrieves only a fraction of total rows due to where clause
- For queries with where clause row estimation depends on **`pg_statistics`**
- The **`pg_statistics`** stores statistics on per columns basis
- Sampling depends on **`default_statistics_target`** configuration parameter (default is 100) it can be changed by

```
ALTER TABLE tablename ALTER COLUMN columnname SET STATISTICS N;
```

Where N is a number between 100 and 10000

Example:

```
ALTER TABLE address ALTER COLUMN phone SET STATISTICS 500;
```

In Above example for phone columns of address table sampling is set to 500.

Extended Statistics

- Planner estimates gets wrong for queries with multiple columns in where clause
- Planner assumes that all columns used in where clause are Independent
- More often columns are correlated

- In such cases we have to use multivariate statistics to help planner
- multivariate stats are not computed automatically but needs to create statistics object

First identify the most frequent queries with multi column where clause or order by clause

Then create statistics object for such columns

Syntax:

```
CREATE STATISTICS creates the statics objects
- CREATE STATISTICS statistics_name
    [ ( statistics_kind ) ]
    ON column_name, column_name [, ...]
    FROM table_name;
```

`statistics_kind` is either `dependencies` or `ndistinct`

Once the object is created **ANALYZE / VACUUM** computes the stats

`pg_statistic_ext` table stores the Extended statistics

Functional Dependencies

- Functional Dependencies exists on Primary Key or Superkey
- However not all databases are 100% Normalized
- Without knowledge of functional dependencies Query planner treats both columns independently which results in inaccurate estimates

Wherever if such functionally dependent columns are used in query's where clause it is better to create statistics object

Syntax

```
CREATE STATISTICS statistics_name   (dependencies)
ON column_name, column_name [, ...]   FROM table_name;
```

Example:

```
CREATE STATISTICS mystat1(dependencies) ON city, country_id FROM city;
```

Run ANALYZE immediately

Functional dependencies are not used for range / like operators

ndistinct

- Its Normal for order by or Group by queries to use multiple columns
- Planner may not get accurate statistics because it examines all involved columns individually
- Such queries need to n-distinct statistics for group of columns

Syntax:

```
CREATE STATISTICS statistics_name  (ndistinct)  ON column_name, column_name [, ...] FROM
table_name;
```

Example:

```
CREATE STATISTICS mystat2(ndistinct)ON first_name, last_name FROM actor;
```

Run ANALYZE immediately

Cost Estimation

- Query Cost Depends many parameters
- Primarily query cost depends on Total rows and total pages in table that can found from pg_class

```
select relpages,reltuples from pg_class where relname='city';
```

Also query cost depends on following parameters

```
Name                  |Setting | Description
----------------------+--------+------------------------------------
random_page_cost      | 4      | Cost of reading a page using index
seq_page_cost         | 1       | Cost of reading a page sequentially
cpu_index_tuple_cost  | 0.005  | Cost of reading a row using Index
cpu_operator_cost     | 0.0025 | Cost of reading a row with WHERE clause
cpu_tuple_cost        | 0.01   | Cost of processing each row
parallel_setup_cost   | 1000   | Cost of setting up of parallel worker
parallel_tuple_cost   | 0.1    | Cost of processing a row parallelly
```

Example1:

```
explain select * from city;
                     QUERY PLAN
--------------------------------------------------------
 Seq Scan on city  (cost=0.00..11.00 rows=600 width=23)
(1 row)
```

It is sequential Scan Cost estimate is as follows

```
select relpages, reltuples from pg_class where relname='city';
 relpages | reltuples
----------+-----------
        5 |       600
```

Cost Calculation

```
Cost =  (relpages * seq_page_cost) + (Estimate Rows * cpu_tuple_cost  )
==> ((5*1)+(600*0.01)) = 11.00
```

Example2:

```
explain select * from city where city like 's%';
                      QUERY PLAN
-----------------------------------------------------
 Seq Scan on city  (cost=0.00..12.50 rows=1 width=23)
   Filter: ((city)::text ~~ 's%'::text)
```

It is a sequential Scan Cost and estimate is as follows

```
select relpages, reltuples from pg_class where relname='city';
```

```
 relpages | reltuples
----------+-----------
        5 |       600
```

Cost Calculation

```
Cost =  (relpages * seq_page_cost) + (Estimate Rows * cpu_tuple_cost  ) + (Estimated Rows
* cpu_operator_cost)
==> ((5*1)+(600*0.01)+(600*0.0025)) = 12.50
```

Tweaking Cost Parameters

- Default values work optimally for most of the workloads
- If changed with same proportion then there is no impact of such change
- `random_page_cost` is always 4 times of `seq_page_cost` as CPU has to read INDEX as well
- However due to less pages over all cost decreases significantly
- Decreasing `random_page_cost` makes planner to consider use of indexes
- Increasing `random_page_cost` will make planner to think that the indexes are more expensive
- If database is 100% cached then set `seq_page_cost` and `random_page_cost` to 1
- If database is heavily cached then set `seq_page_cost` and `random_page_cost` nearer to respective CPU costs parameters
- Custom values for `seq_page_cost` and `random_page_cost` can be set for individual tablespaces
- If the storage is SSD then `random_page_cost` can be set nearer to `seq_page_cost`

Parallel Query Execution

- By default query use one process to execute
- Parallel queries uses more than one process to improve performance
- Planner does not use parallel plan for write queries or any query which locks data
- In Isolation level `serializable` planner ignore parallelism
- Number of processes limited by max_parallel_workers_per_gather

max_parallel_workers_per_gather is limited by max_worker_processes and max_parallel_workers. Also the availability of background workers. In such case `max_worker_processes` and `max_parallel_workers` cab be increased

STEP 4: Restructuring SQL Statements

- Rewriting inefficient SQL is often easier than repairing it
- Avoid implicit type conversion
- Avoid expressions as the optimizer may ignore the indexes on such columns
- Use equi-joins wherever possible to improve SQL efficiency
- Try changing the access path and join orders
- Avoid full table scans if using an index is more efficient

- The join order can have a significant effect on performance
- Use views or materialized views for complex queries

Restructure SQL Statements - Join Order

- Join Order is often determined by Planner
- Sometimes Planner may not select proper join order resulting in cartesian products
- Explicitly use JOIN clause to determine join Order
- Force planner to use join order by setting `join_collapse_limit` to 1

Restructure SQL Statements - Sub Queries

- Sub Queries are often merged with outer query where ever possible
- This results in better sub query performance
- Planner merges the sub queries with outer query if the resulting FROM clause has less than `from_collapse_limit`
- To Control the behavior of Planner adjust this settings

STEP 5: Add / Remove Indexes

- Create indexes as and when needed
- Remove unused indexes
- Adding an index for one SQL can slow down other queries
- Verify index usage using the EXPLAIN command

STEP 6: Review Final Execution Plan

- Check again for missing indexes
- Check table statistics are showing correct estimates
- Check large table sequential scans
- Compare the cost of first and final plans

Memory and Buffers

Global Buffers

Global buffers are allocated at the time startup and retained until PostgreSQL is running.

- Shared Buffers
- WAL Buffers
- Effective Cache Size

Shared Buffers

One of most important configuration setting on a postgres database cluster is **`shared_buffers`**. By setting it optimally we can ensure smooth performance of cluster.

Recommended value for **`shared_buffers`** is 25% installed RAM however we can set it between minimum of 15% to a maximum of 40% physical RAM.

WAL Buffers

Postgres uses **`wal_buffers`** to write the transactions into and them later flushes them to disk files.

Recommended value is this needs to adjusted to roughly 3% **`shared_buffers`**. It can be a maximum of 16MB (WAL file size) to a minimum of 64KB.

effective_cache_size

The **`effective_cache_size`** value provides a 'rough estimate' of the number of how much memory is available for disk caching by the operating system and within the database itself, after taking into account what's used by the OS itself and other applications.

This value is used only by the PostgreSQL query planner to figure out whether plans it's considering would be expected to fit in RAM or not. If it's set too low, indexes may not be used for executing queries the way you'd expect

Ideal setting will be 50% physical RAM and this values is independent of **`shared_buffers`**.

Per Session Buffers

Before setting these parameters consider **`max_connections`** as these buffers may be allocated to each session as required.

work_mem

This parameter specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. If a lot of complex sorts are happening, and you have enough memory, then increasing the **`work_mem`** parameter allows PostgreSQL to do larger in-memory sorts which will be faster than disk based equivalents.

Normal Recommended Value

```
work_mem = Total RAM * 0.25 / max_connections
```

How to set it

- Enable **`log_temp_files`** parameter and then look for lines containing 'temporary file'.
- Note the file size of temporary file
- Set the **`work_mem`** to double the size of temporary file

maintenance _work_mem

This parameter specifies the maximum amount of memory used by maintenance operations such as **`VACUUM`**, **`CREATE INDEX`** and **`ALTER TABLE ADD FOREIGN KEY`**. As these operation are not so frequent this can be set to a bit higher value however consider the **`autovacuum_max_workers`** as it multiplies when auto vacuum is running.

Temp_buffers

PostgreSQL database utilizes this memory area for holding the temporary tables of each session, these will be cleared when the connection is closed.

calculate RAM (memory) recommendation

RAM recommendation for PostgreSQL database cluster can be done using the formula below.

```
RAM_REQUIRED =  (SumOfGlobalBuffers  +  (MaxConnections * SumOfSessionBuffers) +
TotalAutovacuumWorkMem)
```

Where

```
SumOfGlobalBuffers = shared_buffers + wal_buffers
SumOfSessionBuffers  =  work_mem +  temp_buffers
TotalAutovacuumWorkMem = autovacuum_max_workers * maintenance_work_mem
MaxConnections  =  max_connections
```

Appendix A

PostgreSQL Error Codes

Error Code	Condition Name
Class 00 — Successful Completion	
00000	successful_completion
Class 01 — Warning	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)	
02000	no_data
02001	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete	
03000	sql_statement_not_yet_complete
Class 08 — Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Class 09 — Triggered Action Exception	
09000	triggered_action_exception
Class 0A — Feature Not Supported	
0A000	feature_not_supported
Class 0B — Invalid Transaction Initiation	
0B000	invalid_transaction_initiation

Error Code	Condition Name
Class 0F — Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L — Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P — Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z — Diagnostics Exception	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 — Case Not Found	
20000	case_not_found
Class 21 — Cardinality Violation	
21000	cardinality_violation
Class 22 — Data Exception	
22000	data_exception
2202E	array_subscript_error
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet

Error Code	Condition Name
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
22013	invalid_preceding_or_following_size
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
2200H	sequence_generator_limit_exceeded
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
22030	duplicate_json_object_key_value

Error Code	Condition Name
22031	invalid_argument_for_sql_json_datetime_function
22032	invalid_json_text
22033	invalid_sql_json_subscript
22034	more_than_one_sql_json_item
22035	no_sql_json_item
22036	non_numeric_sql_json_item
22037	non_unique_keys_in_a_json_object
22038	singleton_sql_json_item_required
22039	sql_json_array_not_found
2203A	sql_json_member_not_found
2203B	sql_json_number_not_found
2203C	sql_json_object_not_found
2203D	too_many_json_array_elements
2203E	too_many_json_object_members
2203F	sql_json_scalar_required
2203G	sql_json_item_cannot_be_cast_to_target_type
Class 23 — Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 — Invalid Cursor State	
24000	invalid_cursor_state
Class 25 — Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transaction
25005	no_active_sql_transaction_for_branch_transaction

Error Code	Condition Name
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
25P03	idle_in_transaction_session_timeout
25P04	transaction_timeout
Class 26 — Invalid SQL Statement Name	
26000	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 — Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B — Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F — SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 — External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception	

Error Code	Condition Name
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3B — Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D — Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F — Invalid Schema Name	
3F000	invalid_schema_name
Class 40 — Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch

Error Code	Condition Name
42P22	indeterminate_collation
42809	wrong_object_type
428C9	generated_always
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation	
44000	with_check_option_violation
Class 53 — Insufficient Resources	
53000	insufficient_resources

Error Code	Condition Name
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 — Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Class 55 — Object Not In Prerequisite State	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
55P04	unsafe_new_enum_value_usage
Class 57 — Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped
57P05	idle_session_timeout
Class 58 — System Error (errors external to PostgreSQL itself)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Class F0 — Configuration File Error	
F0000	config_file_error
F0001	lock_file_exists
Class HV — Foreign Data Wrapper Error (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found

Error Code	Condition Name
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection
Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Class XX — Internal Error	
XX000	internal_error
XX001	data_corrupted

Error Code	Condition Name
XX002	index_corrupted