

Application Delivery Fundamentals 2.0 B:

Introduction to Postgres

Parameswari Ettiappan

Postgres Goals

- History Of Postgre SQL
- Major Features
- Connecting to Postgre SQL DB using psql
- Connecting to Pgadmin tool
- Writing Simple Queries In Postgre
- Understanding Data types
- Understanding Single row Functions in PostgreSQL
- Specific features of SQL in PostgreSQL as compared to Oracle Database

Postgres Goals

- Installing and Configuring Postgre in Windows OS
- Understanding Data types in Postgre
- New Data type data types of Postgre
- Single row functions of Postgre
- Understanding Group Functions and writing Queries (Postgre Specific)

Postgres Goals

- Understanding DDL's in Postgre
- Creating Tables/Constraints
- Understanding DML's and MVCC in Postgre
- Creating Other Database Objects in Postgre
- Views/Materialized View/Indexes/sequences
- Btree/Partial Indexes In Postgre
- Understanding Query Performance issues in Postgre
- Explain Vs Explain Analyze options and interpretation
- Table Statistics

Postgres Goals

- Moving Data the Copy command
- Querying Metadata Information Postgre
- Compare Dictionaries Vs Meta data tables of Postgre
- Table Partitions In Postgre
- Partition Setup

Postgres Goals

- Introduction to PL/PGSQL
- Control structures Overview
- Loops Overview
- Creating Anonymous blocks in postgre
- Understanding Cursors
- Bound Vs Unbound Cursors
- Creating Subprograms (Stored Procedures/Functions) in PLPGSQL
- Exceptions in Postgres
- Database Triggers in Postgre
- Writing a Function as Trigger body
- Basic Architecture of a Postgres DB from the perspective of a developer

What is PostgreSQL

- PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES,
- Version 4.21, 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.

What is PostgreSQL

- 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
 - pdatable views
 - transactional integrity
 - multiversion concurrency control

What is PostgreSQL

- PostgreSQL can be extended by the user in many ways, for example by adding new
 - data types
 - functions
 - operators
 - aggregate functions
 - index methods
 - procedural languages

Postgre History

- The object-relational database management system now known as PostgreSQL is derived from the POSTGRES 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.

Postgre History

- The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.
- The implementation of **POSTGRES** began in 1986.
- In 1994, Andrew Yu and Jolly Chen added an SQL language interpreter to POSTGRES.
- Under a new name, **Postgres95** was subsequently released to the web to find its own way in the world as an opensource descendant of the original POSTGRES Berkeley code.

Postgre History

- By 1996, it became clear that the name “Postgres95” would not stand the test of time.
- We chose a new name, **PostgreSQL**, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability.
- At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Berkeley POSTGRES project.

Installation

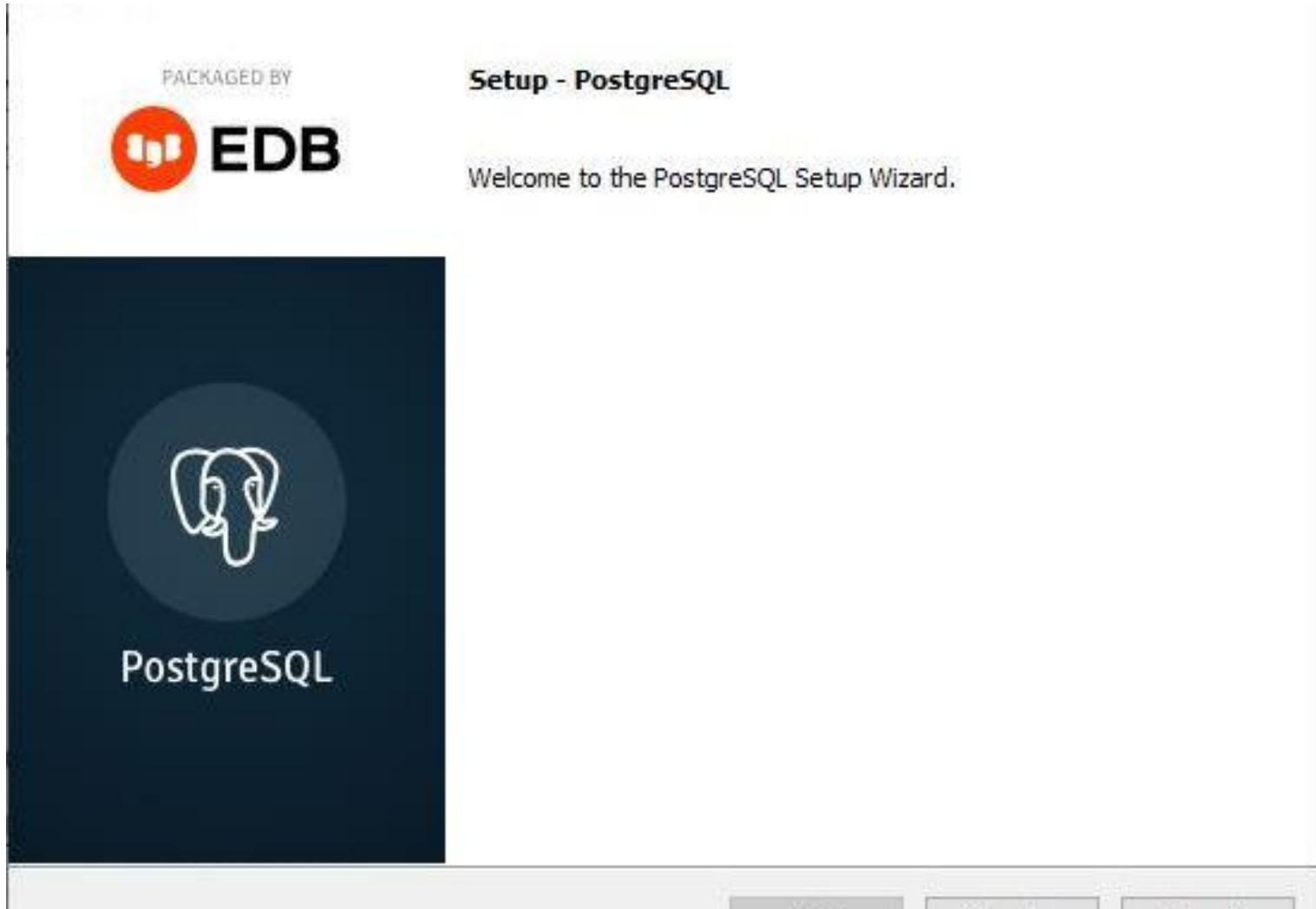
- Download PostgreSQL Installer for Windows
- Install PostgreSQL on Window step by step.
- Verify the Installation

Installation

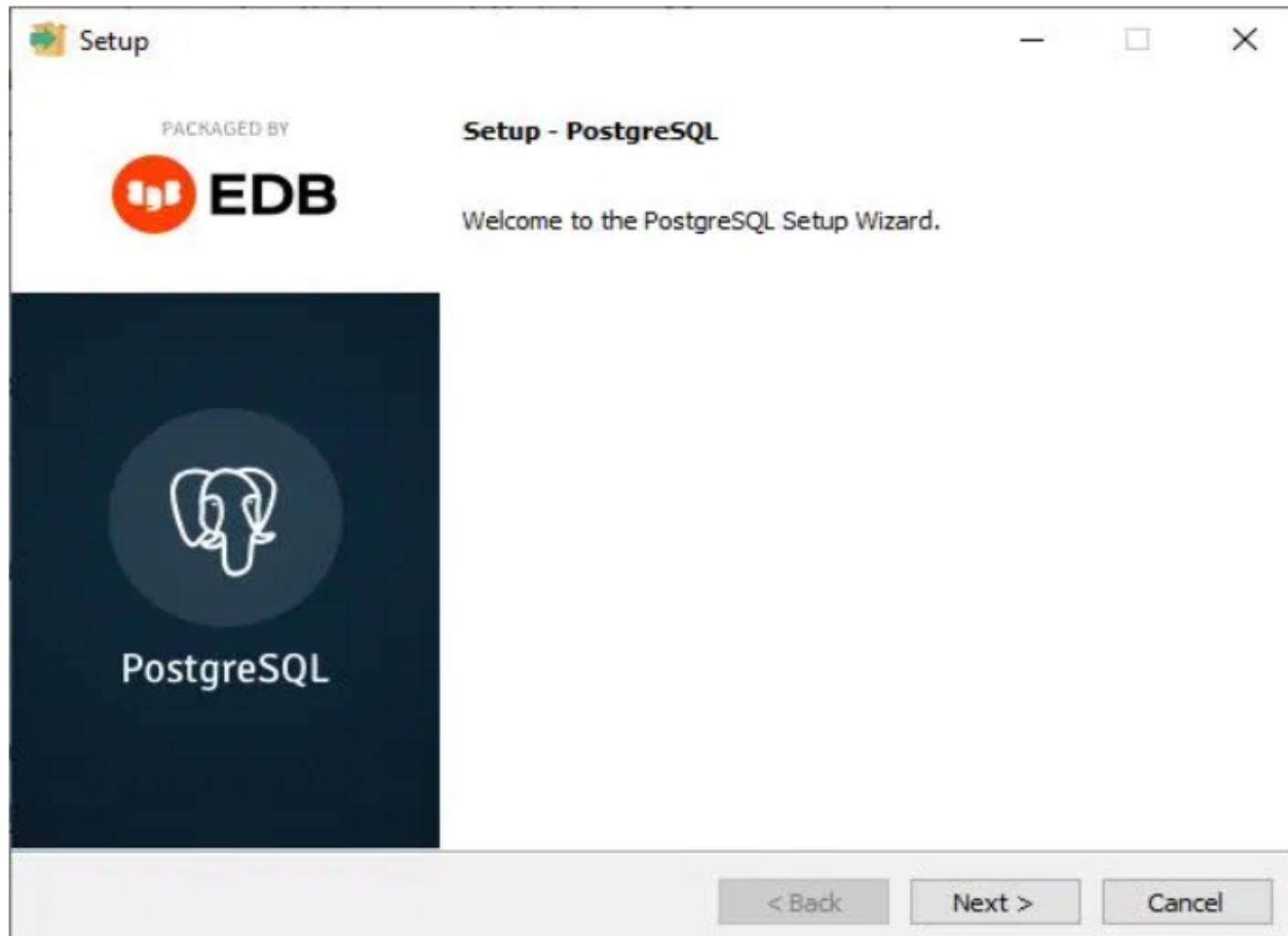
Access the [POSTGRESQL](#) website and download the POSTGRESQL installation package.

Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
13	N/A	N/A	Download	Download	N/A
12.4	N/A	N/A	Download	Download	N/A
11.9	N/A	N/A	Download	Download	N/A
10.14	Download				
9.6.19	Download				
9.5.23	Download				
9.4.26 (Not Supported)	Download				
9.3.25 (Not Supported)	Download				

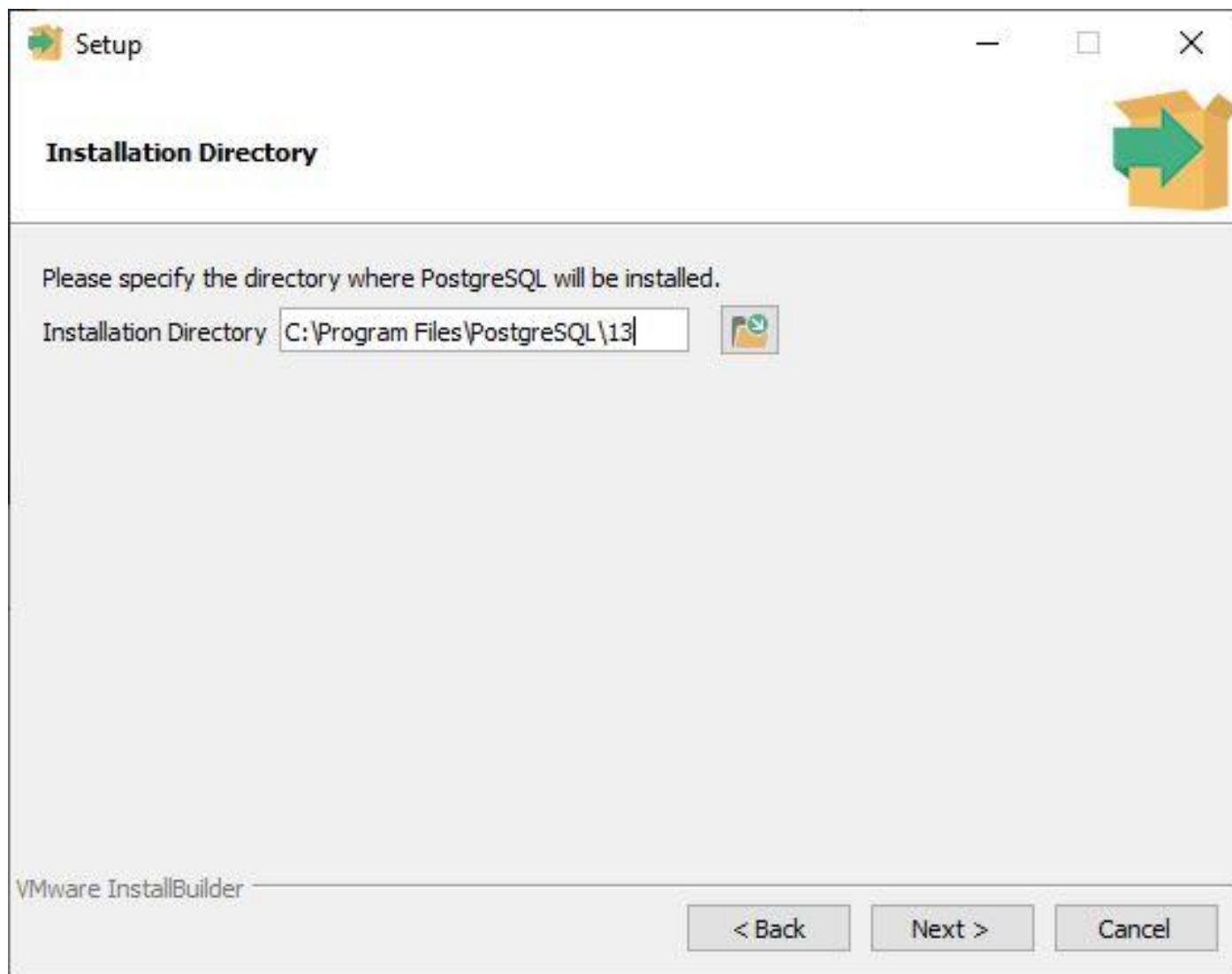
Installation



Installation



Installation



Installation

Setup

Select Components



Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

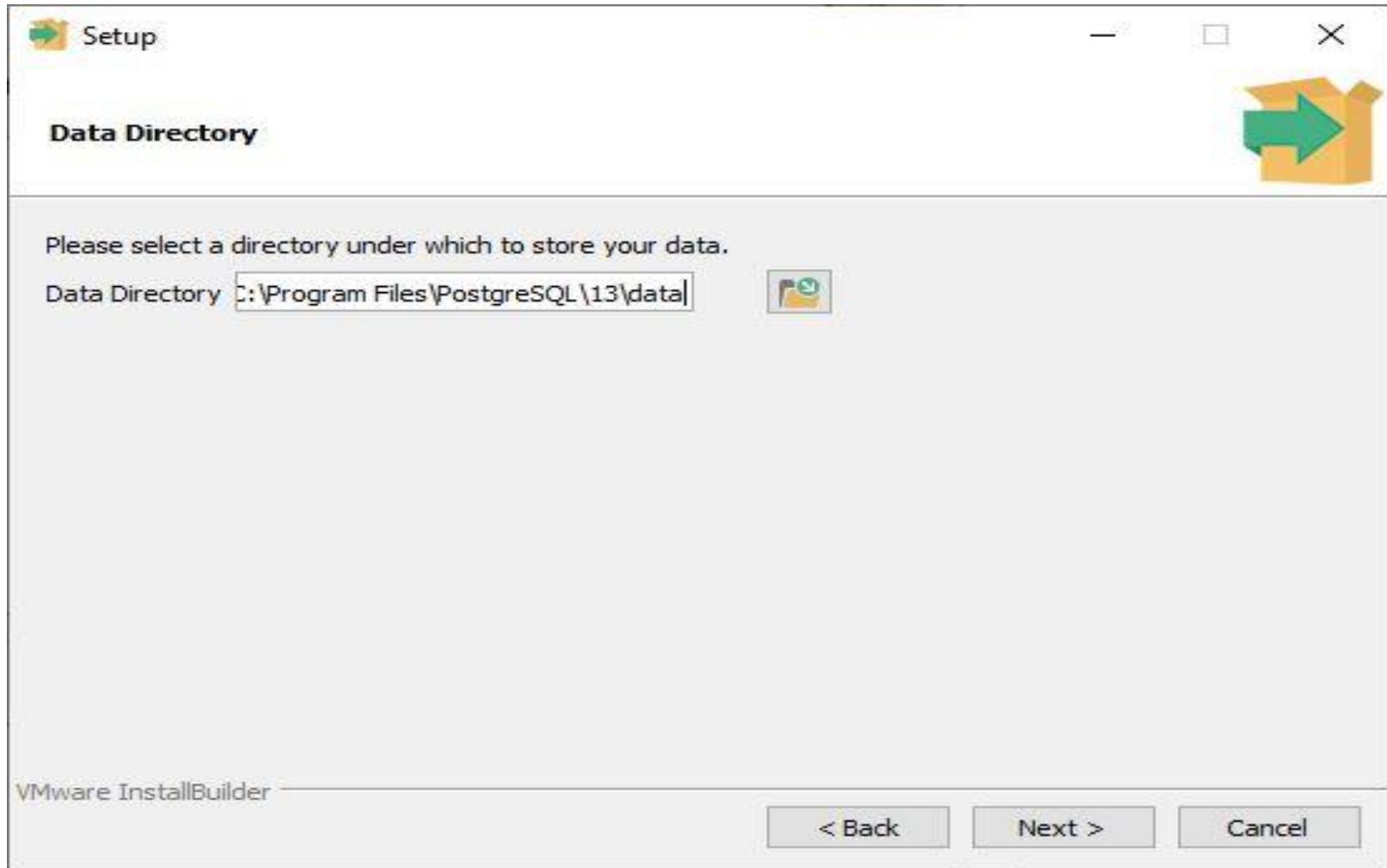
PostgreSQL Server
 pgAdmin 4
 Stack Builder
 Command Line Tools

Click on a component to get a detailed description

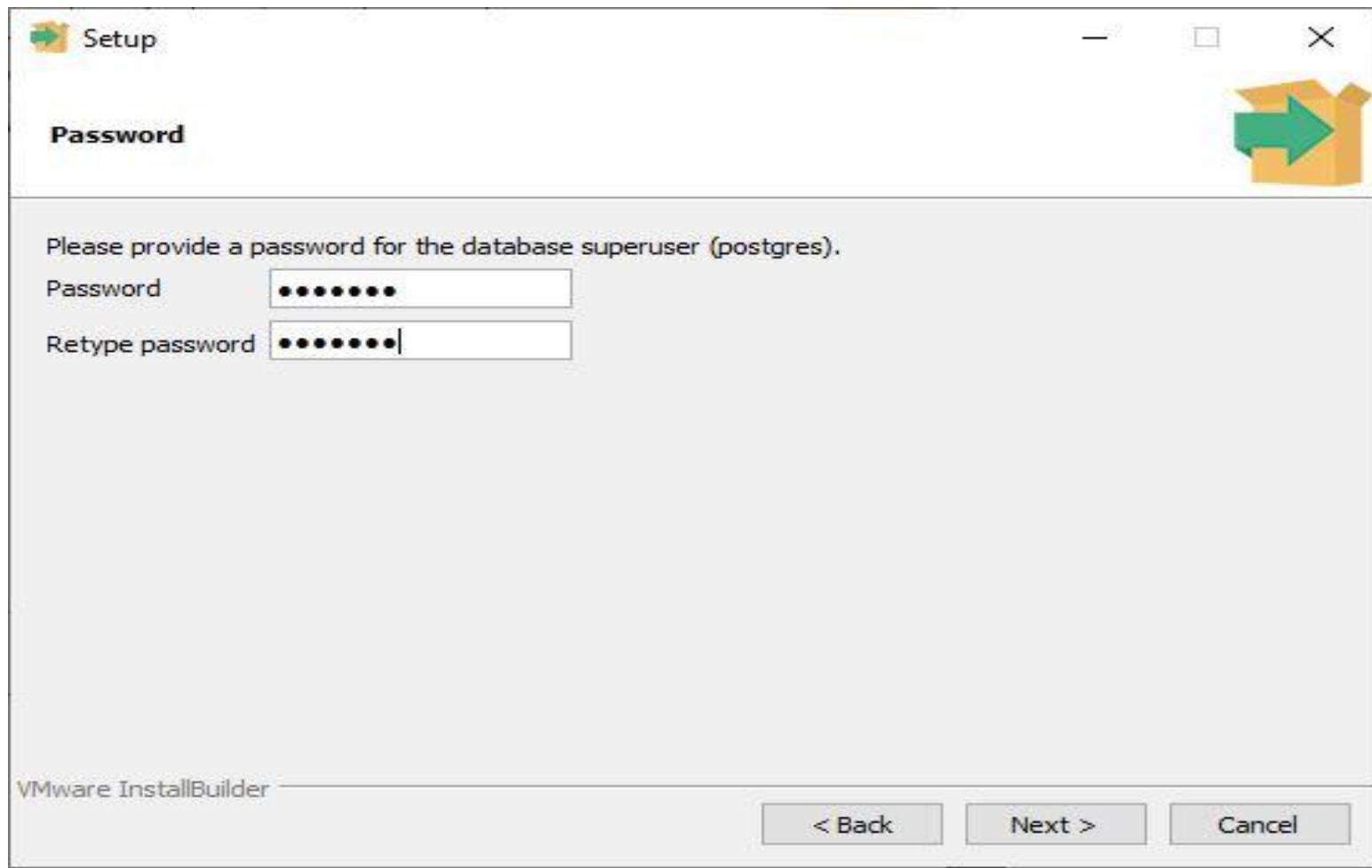
VMware InstallBuilder

< Back Next > Cancel

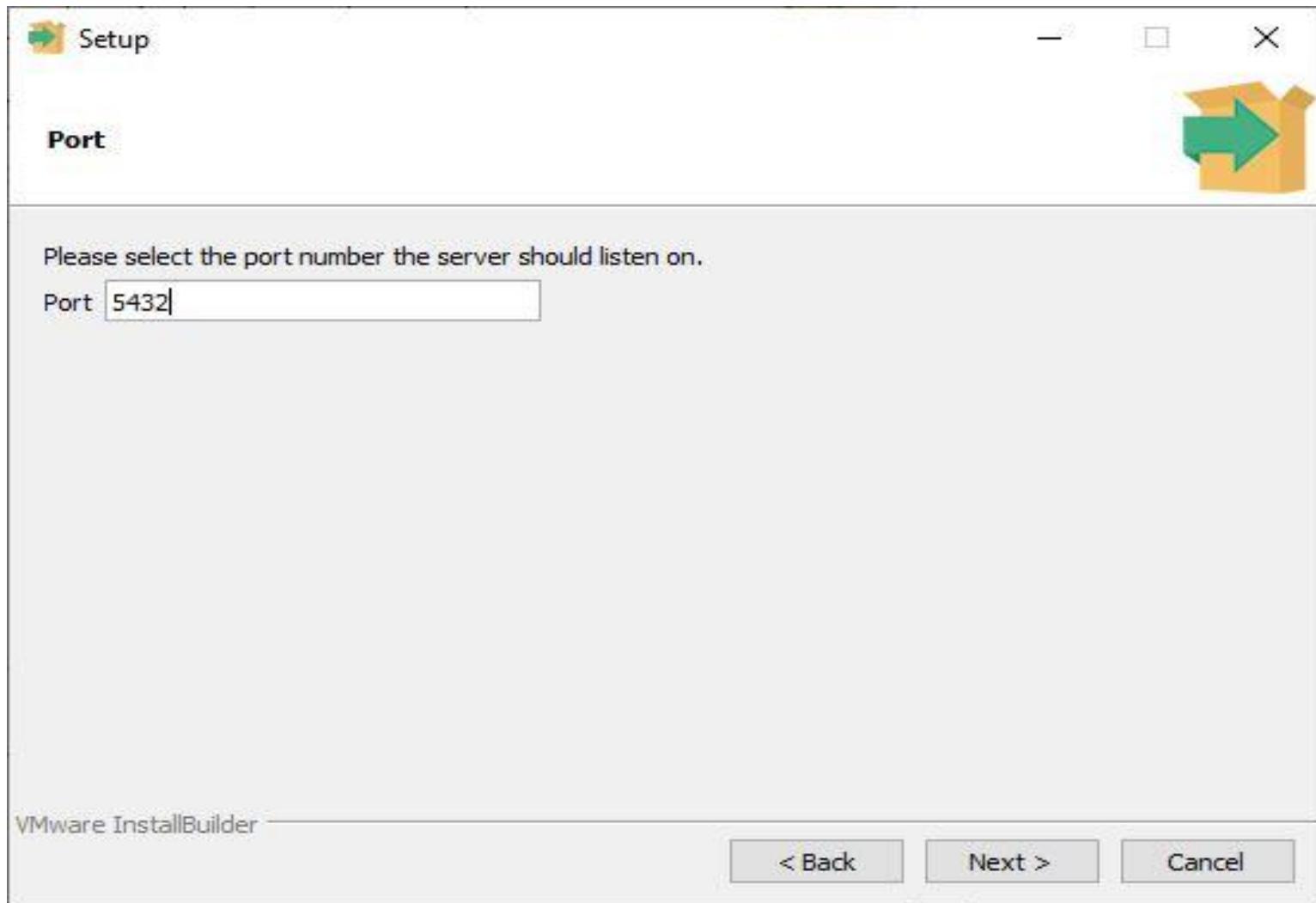
Installation



Installation



Installation



Installation

Setup

Advanced Options



Select the locale to be used by the new database cluster.

Locale [Default locale] ▾

VMware InstallBuilder

< Back Next > Cancel

Installation

Setup

Pre Installation Summary



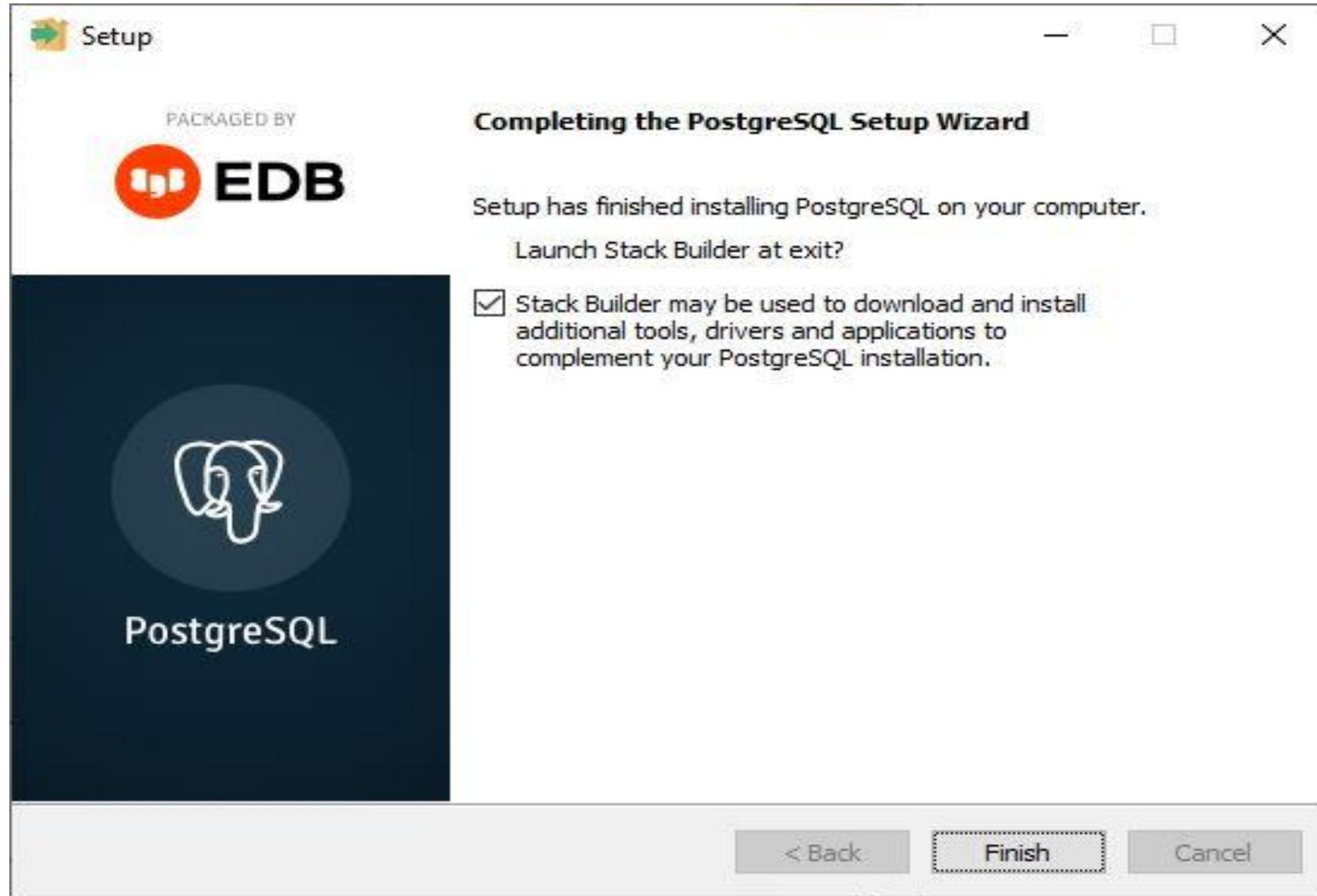
The following settings will be used for the installation::

```
Installation Directory: C:\Program Files\PostgreSQL\13
Server Installation Directory: C:\Program Files\PostgreSQL\13
Data Directory: C:\Program Files\PostgreSQL\13\data
Database Port: 5432
Database Superuser: postgres
Operating System Account: NT AUTHORITY\NetworkService
Database Service: postgresql-x64-13
Command Line Tools Installation Directory: C:\Program Files\PostgreSQL\13
pgAdmin4 Installation Directory: C:\Program Files\PostgreSQL\13\pgAdmin 4
Stack Builder Installation Directory: C:\Program Files\PostgreSQL\13
```

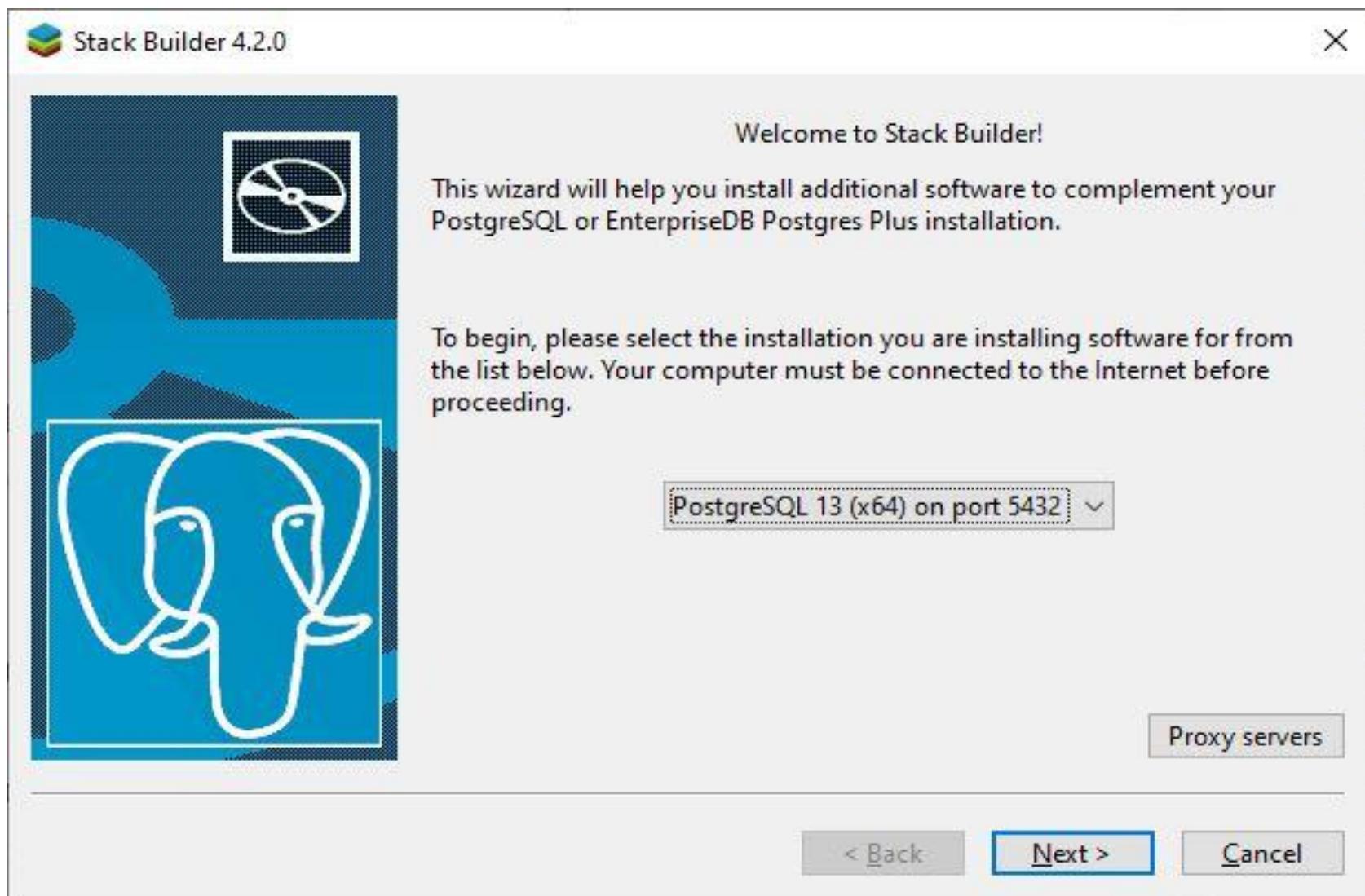
VMware InstallBuilder

< Back Next > Cancel

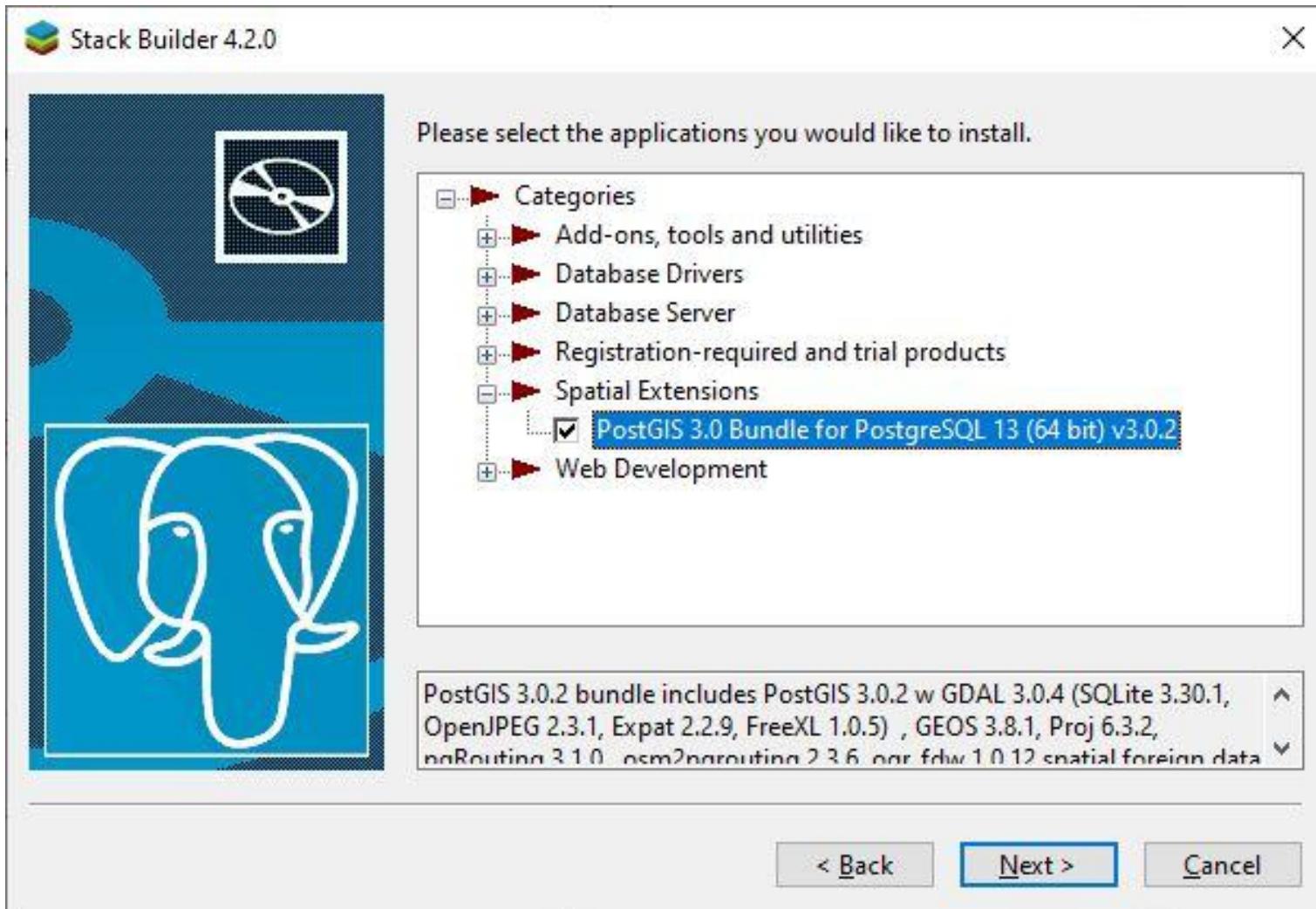
Installation



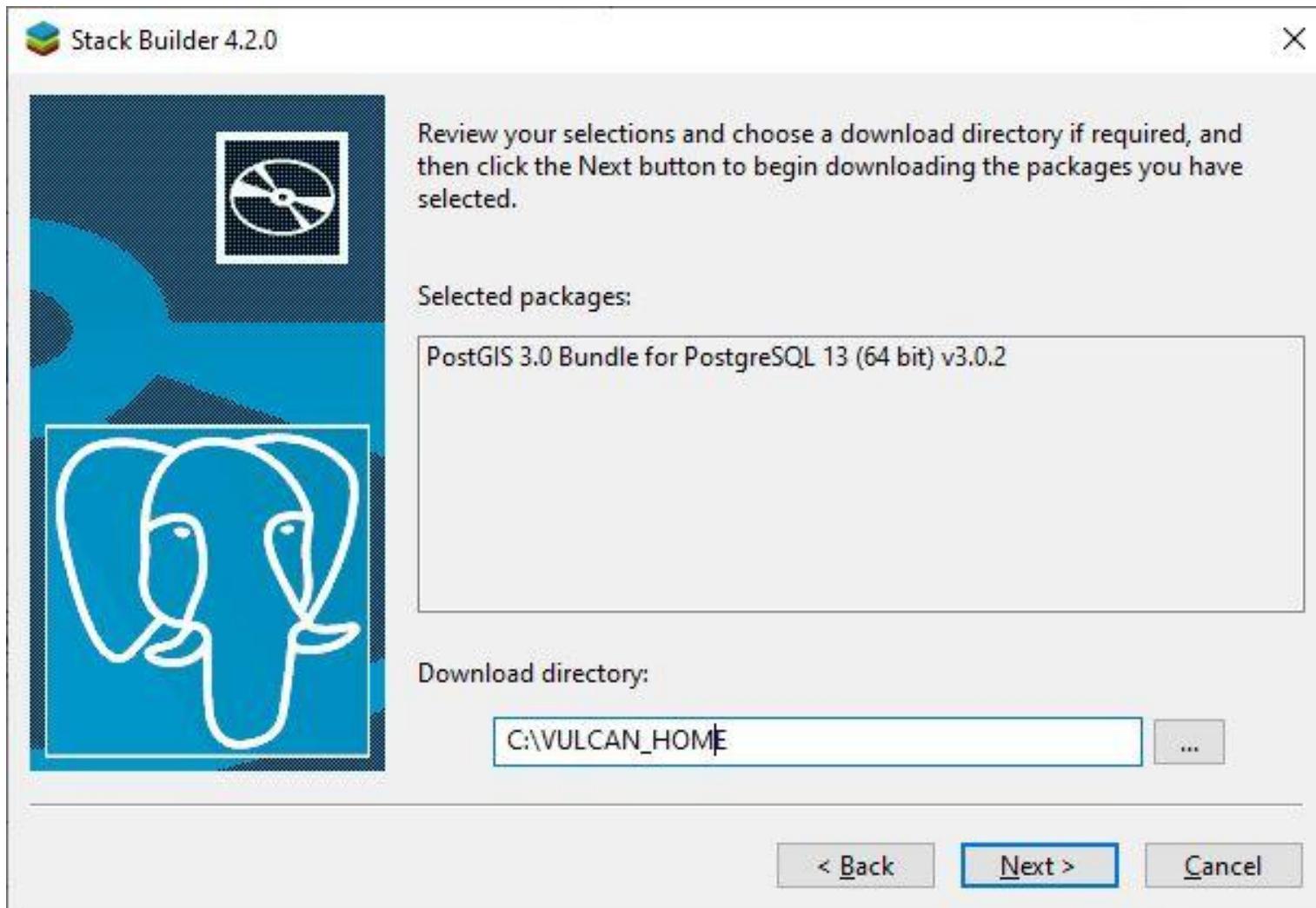
Install PostGIS 3.0 on PostgreSQL 13



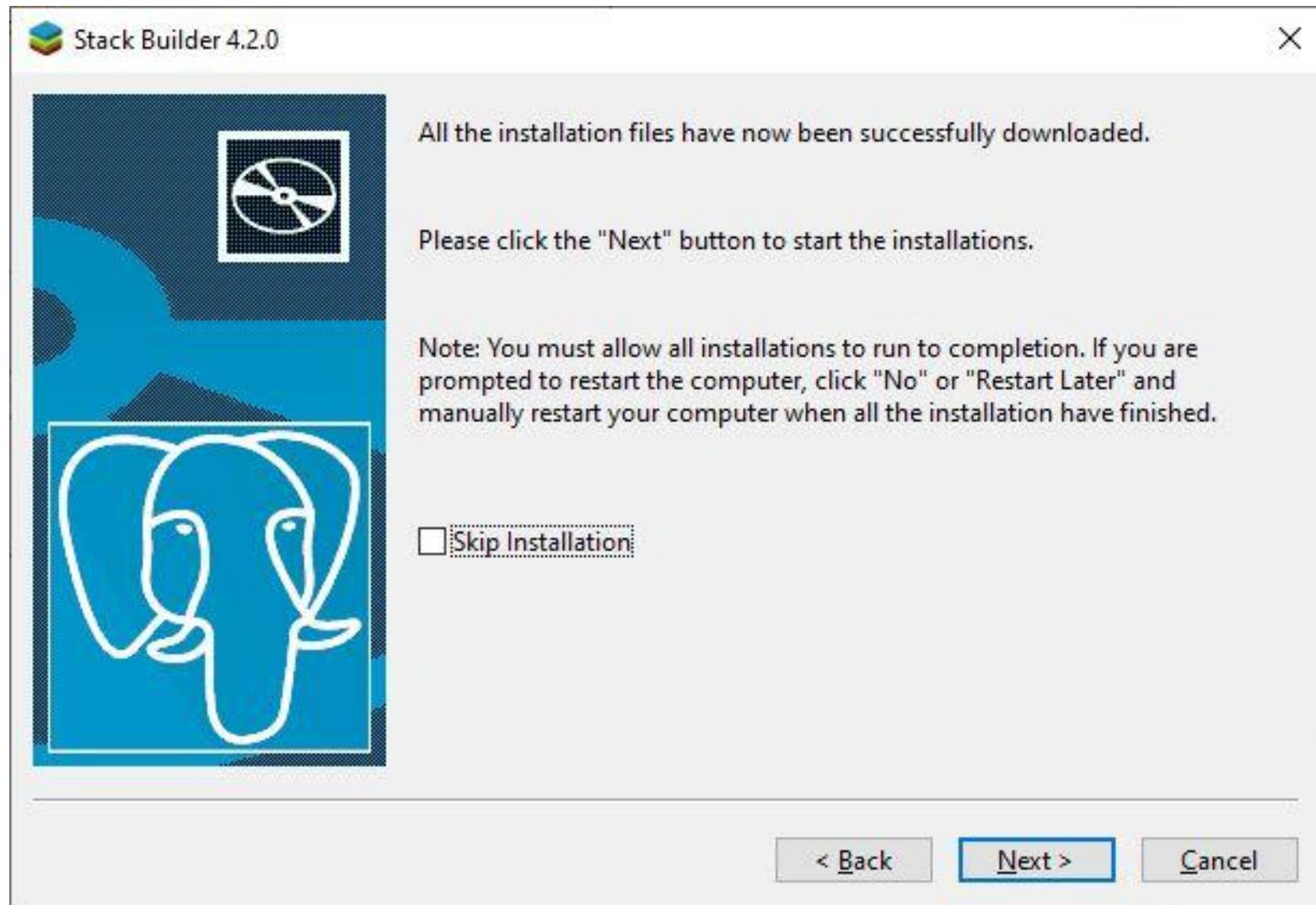
Install PostGIS 3.0 on PostgreSQL 13



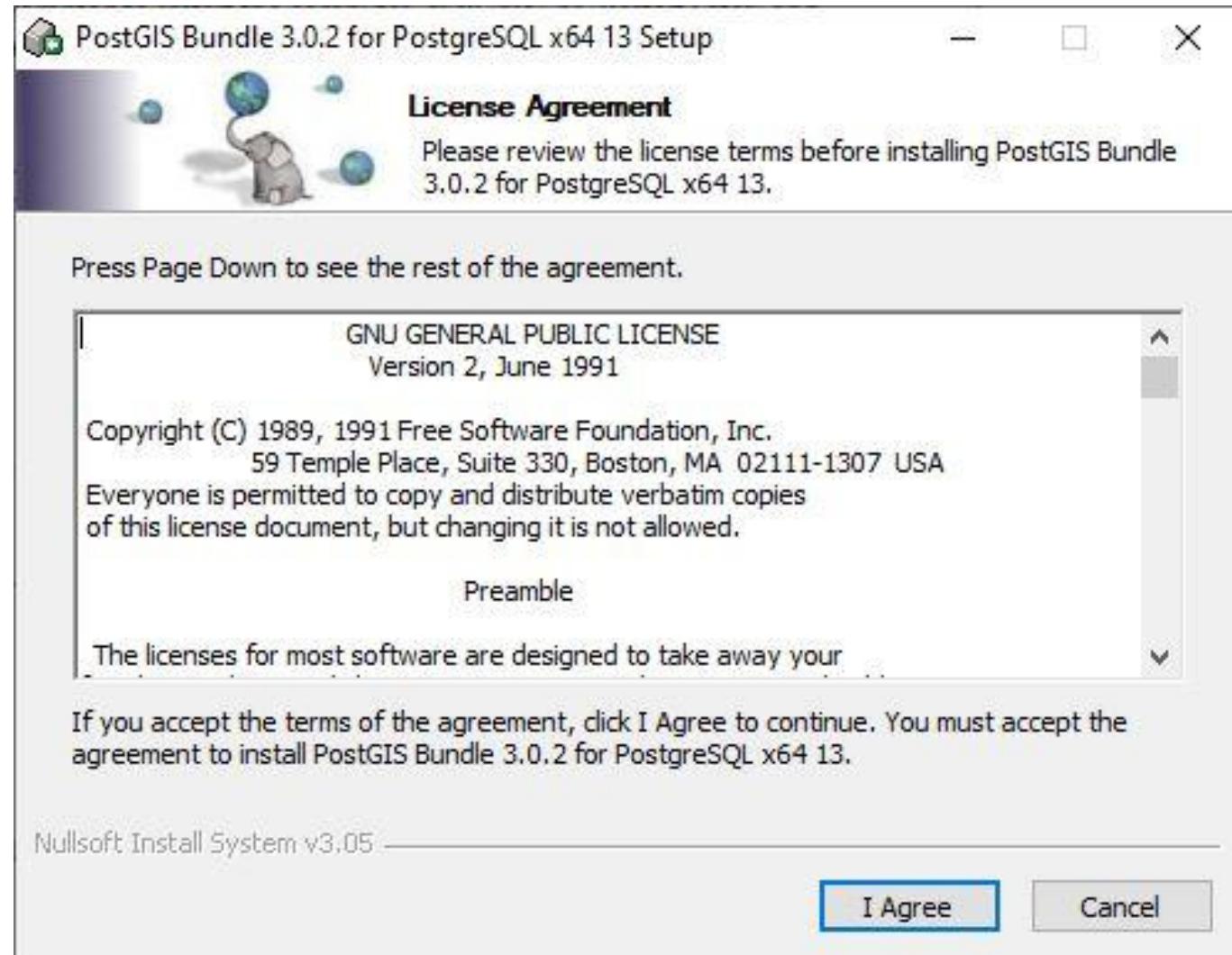
Install PostGIS 3.0 on PostgreSQL 13



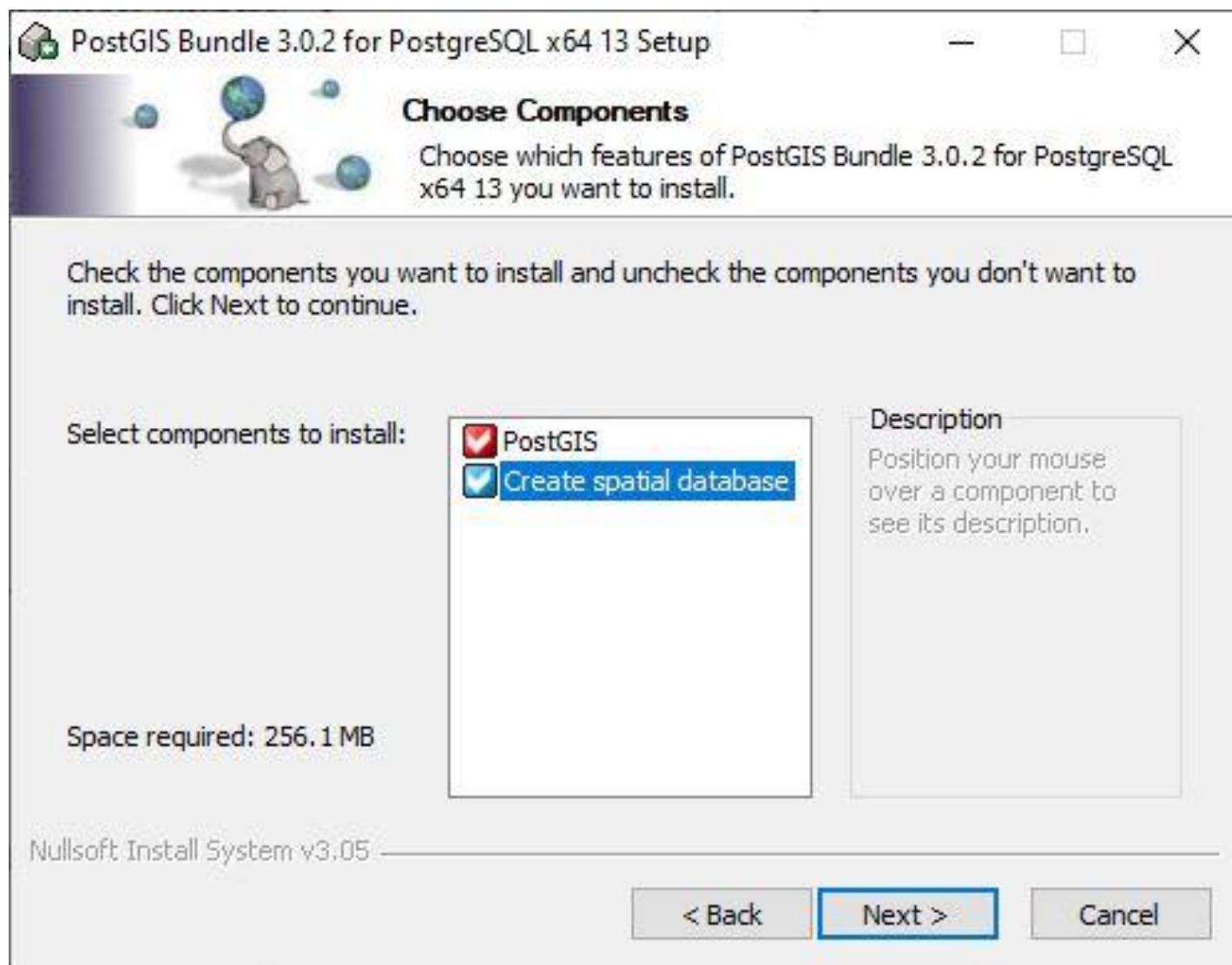
Install PostGIS 3.0 on PostgreSQL 13



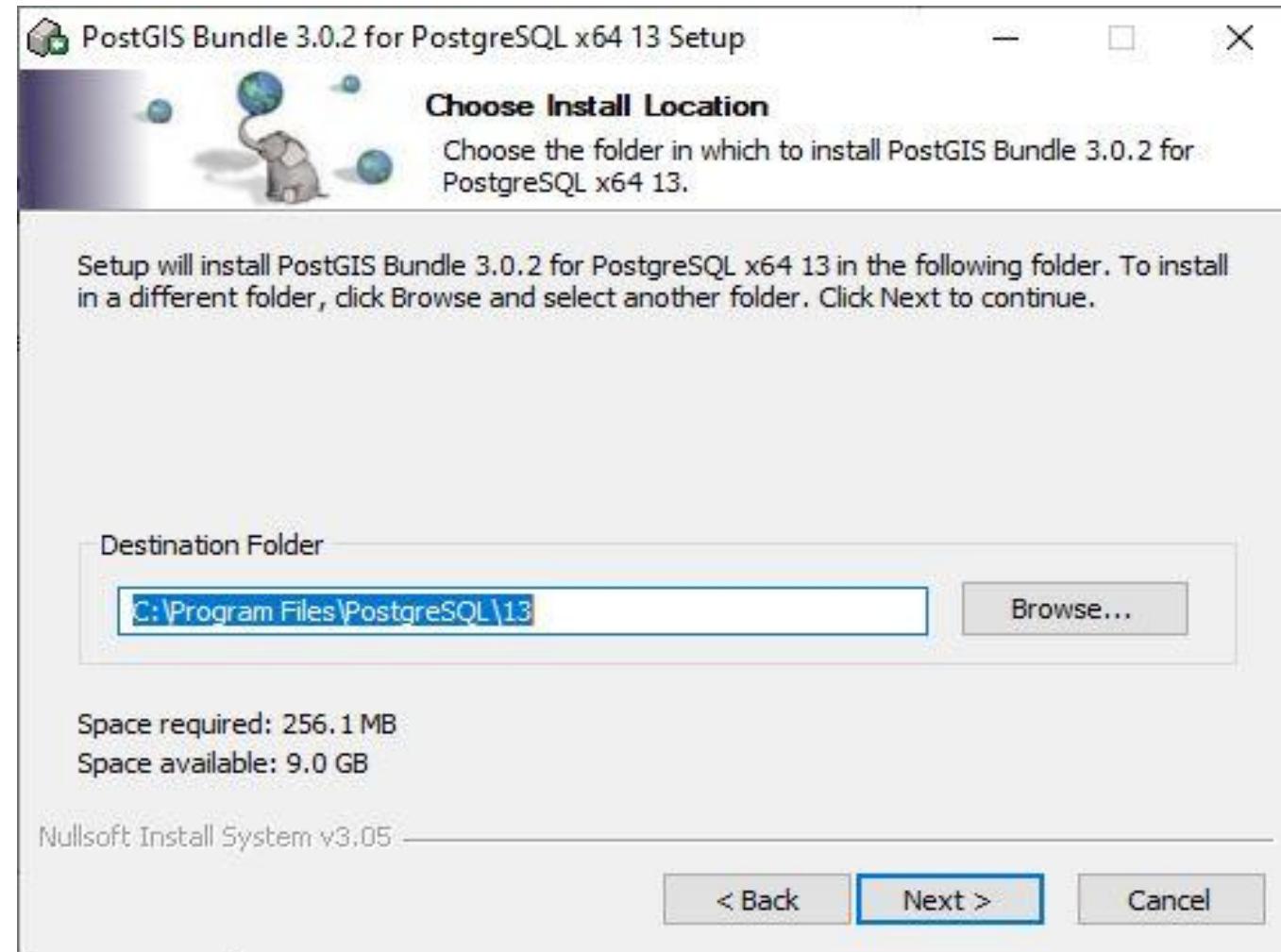
Install PostGIS 3.0 on PostgreSQL 13



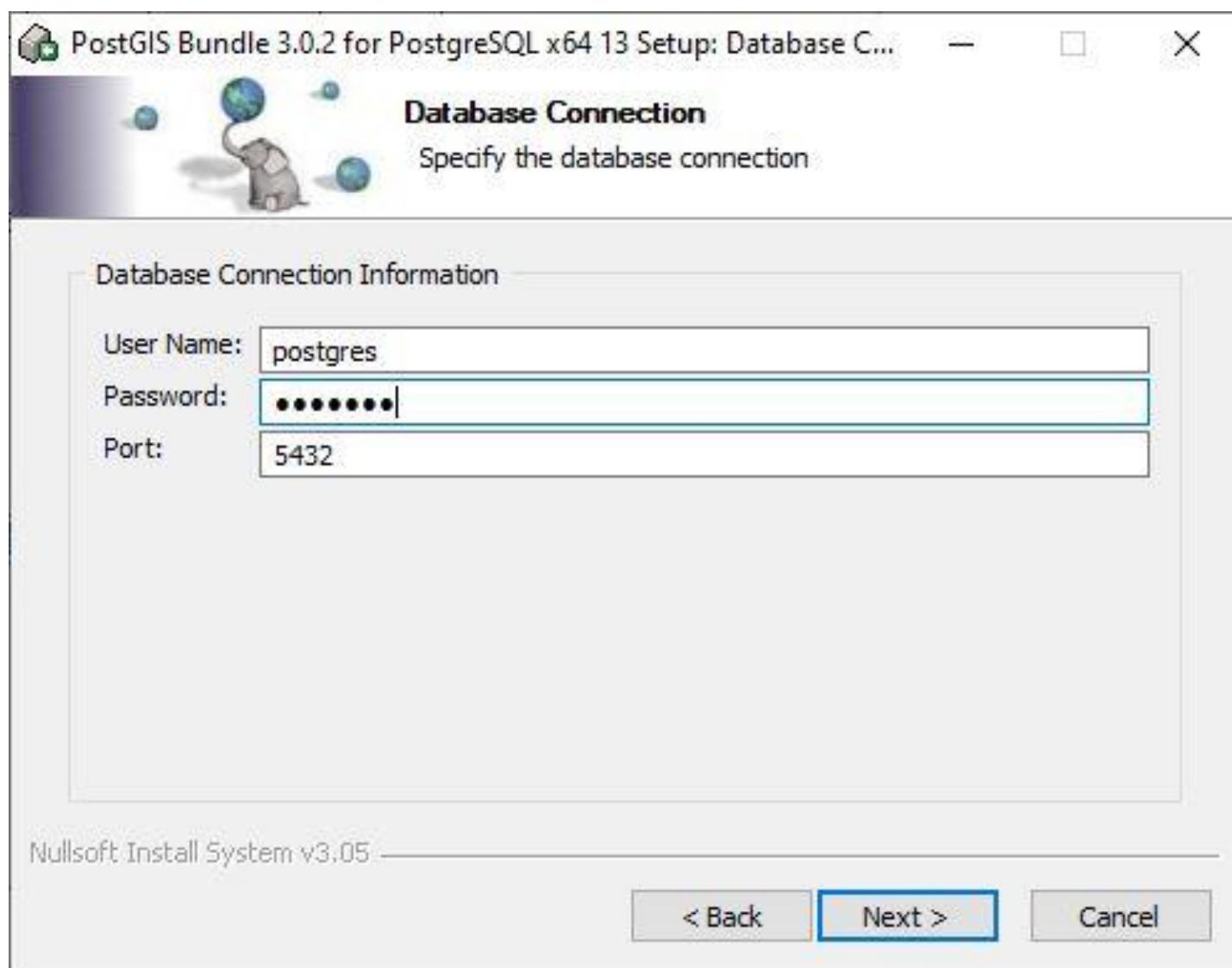
Install PostGIS 3.0 on PostgreSQL 13



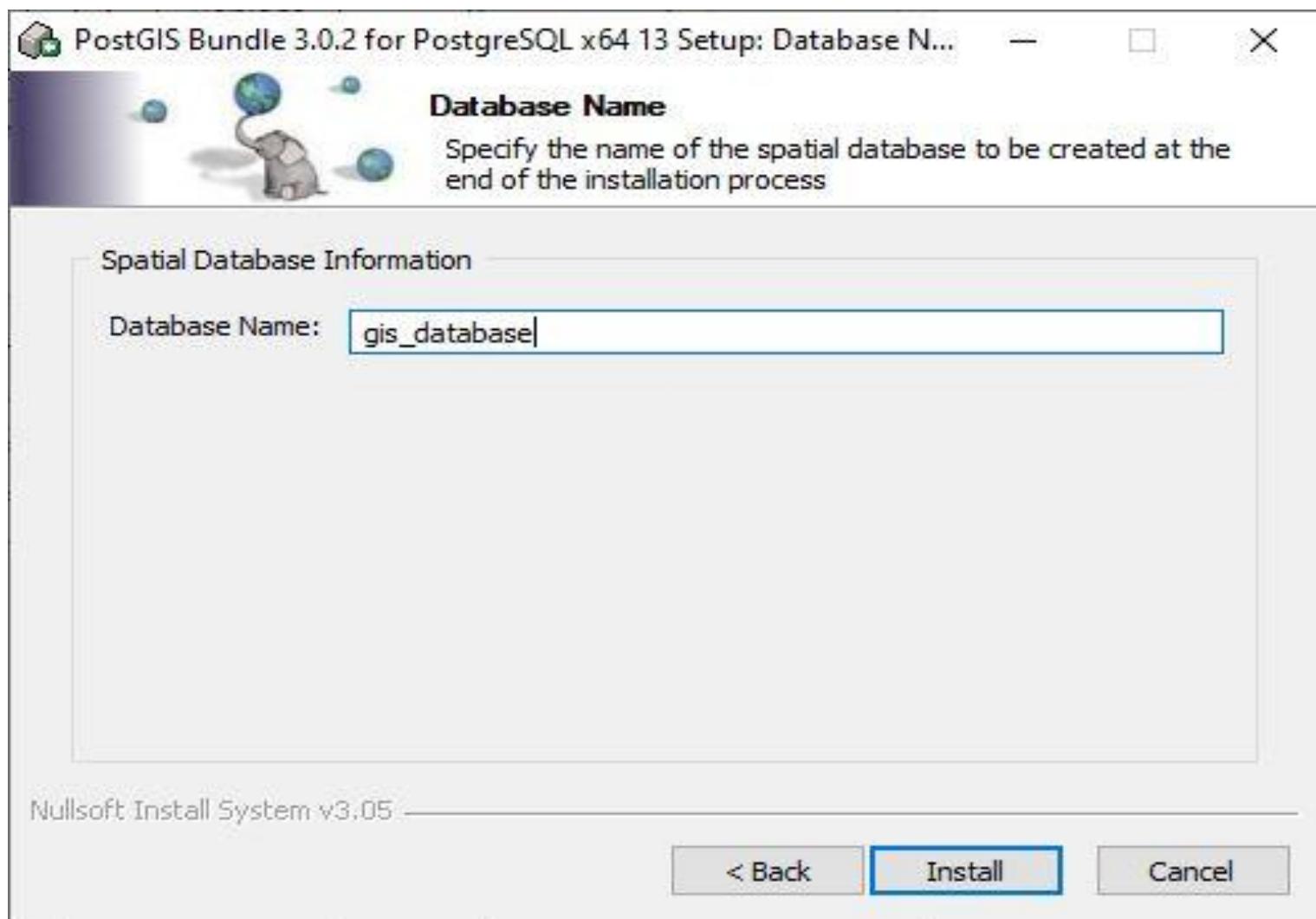
Install PostGIS 3.0 on PostgreSQL 13



Install PostGIS 3.0 on PostgreSQL 13



Install PostGIS 3.0 on PostgreSQL 13



Ubuntu Postgre Installation

- Step 1: Update Package List
 - **`sudo apt update`**
- Step 2: Install PostgreSQL and Contrib Package
 - **`sudo apt install postgresql postgresql-contrib -y`**
- Step 3: Check PostgreSQL Service Status
 - **`sudo systemctl status postgresql`**
- To start/stop manually:
 - **`sudo systemctl start postgresql`**
 - **`sudo systemctl stop postgresql`**

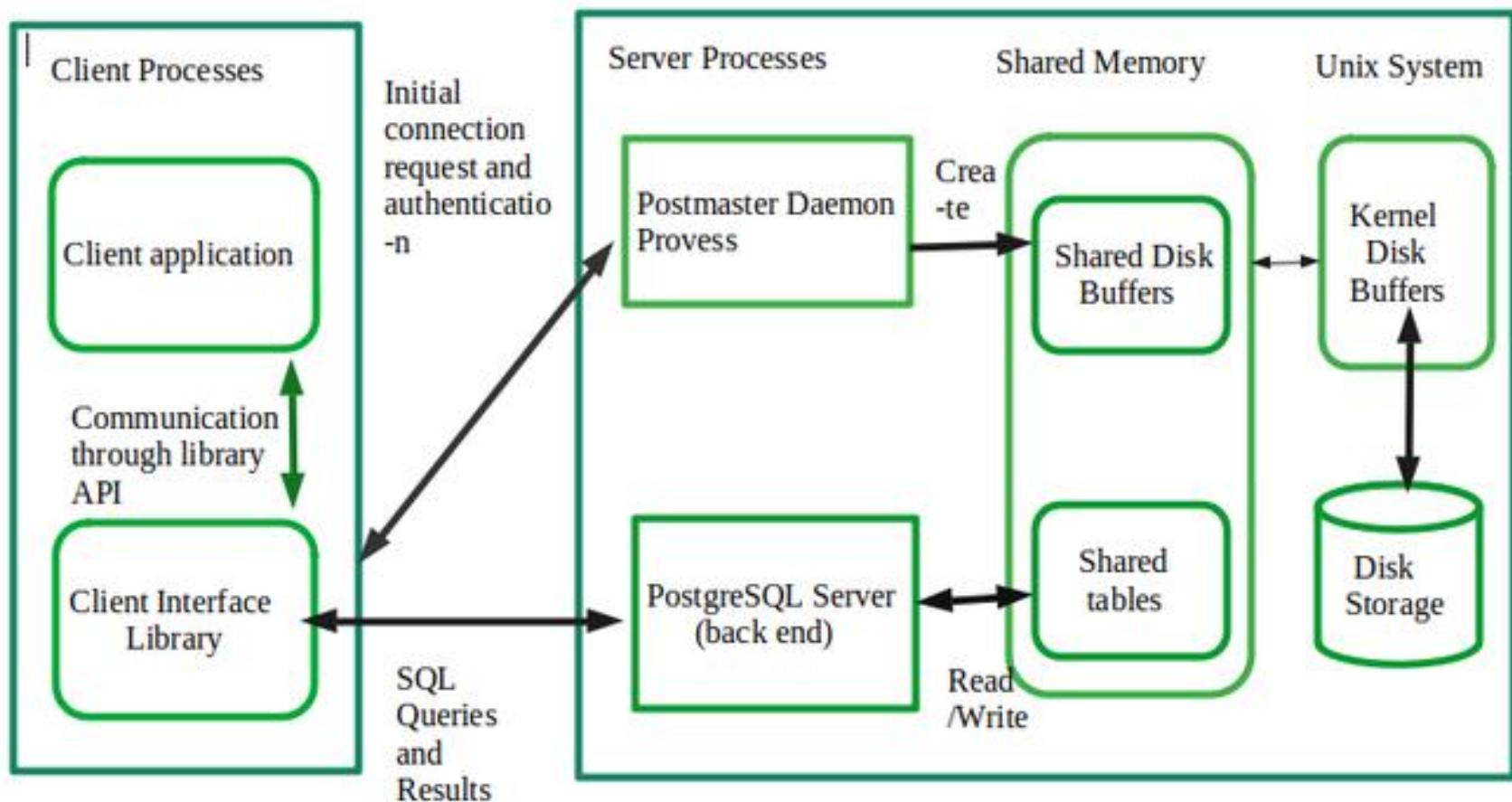
Ubuntu Postgre Installation

- Step 4: Switch to the postgres User
 - **sudo -i -u postgres**
- Then access PostgreSQL shell:
 - **psql**

Postgre Architecture

- PostgreSQL has a Client-server model of architecture. In the simplest term a PostgreSQL service has 2 processes:
- Server-side process: This is the “Postgres” application that manages connections, operations, and static & dynamic assets.
- Client-side process(Front-end applications): These are the applications that users use to interact with the database.
- It generally has a simple UI and is used to communicate between the user and the database generally through APIs.

Client side Process



Core Components

- **1. Postmaster (Main Process)**
 - Initializes the database server.
 - Manages incoming connections.
 - Forks a new **backend process** for each client.

Core Components

- **2. Client Backend Processes**
 - Dedicated process for each connected client.
 - Handles SQL query parsing, planning, execution.
 - Communicates with client applications.

Core Components

- **3. Parser, Planner, Executor**
 - **Parser**: Parses SQL into internal parse trees.
 - **Planner/Optimizer**: Chooses the best execution strategy.
 - **Executor**: Executes the query and returns results.

Core Components

- **4. Shared Memory / Buffers**
 - **Shared Buffers:** Cache for data blocks to reduce disk I/O.
 - **WAL Buffers:** Cache for Write-Ahead Log entries before flush.
 - **Work Mem:** Used for sorts, joins, and hashing.

Core Components

- **5. System Catalogs**
 - Metadata about tables, indexes, roles, etc.
 - Stored in special tables (pg_class, pg_attribute, etc.).

Core Components

- **6. Access Methods**
 - Interface to read/write table (heap) and index data.
 - Supported index types: B-tree, Hash, GiST, GIN, BRIN, etc.

Core Components

- **7. Storage Layer**
 - Each table/index is stored as multiple 8KB pages.
 - WAL ensures durability (Write-Ahead Logging).

Background Processes

Process	Purpose
WAL Writer	Flushes WAL from memory to disk
Checkpointer	Writes dirty buffers to disk at checkpoints
Autovacuum	Reclaims space and updates stats
Archiver	Archives WAL files if enabled
Stats Collector	Gathers usage stats for optimizer
Logical Replication Launcher	Starts logical replication workers

WAL (Write-Ahead Logging)

- **Ensures Durability and Crash Recovery.**
 - Every change is first logged in WAL before modifying data files.
 - Used for replication and PITR (Point In Time Recovery).

Optional Features

- Replication/Streaming: Standby servers can replicate WAL.
- Extensions: PostGIS, pg_stat_statements, uuid-ossp.
- Logical Decoding: Enables streaming changes to external systems.

Useful Configuration Files

- `postgresql.conf` – Main configuration file.
- `pg_hba.conf` – Client authentication rules.
- `pg_ident.conf` – Maps OS usernames to DB roles.

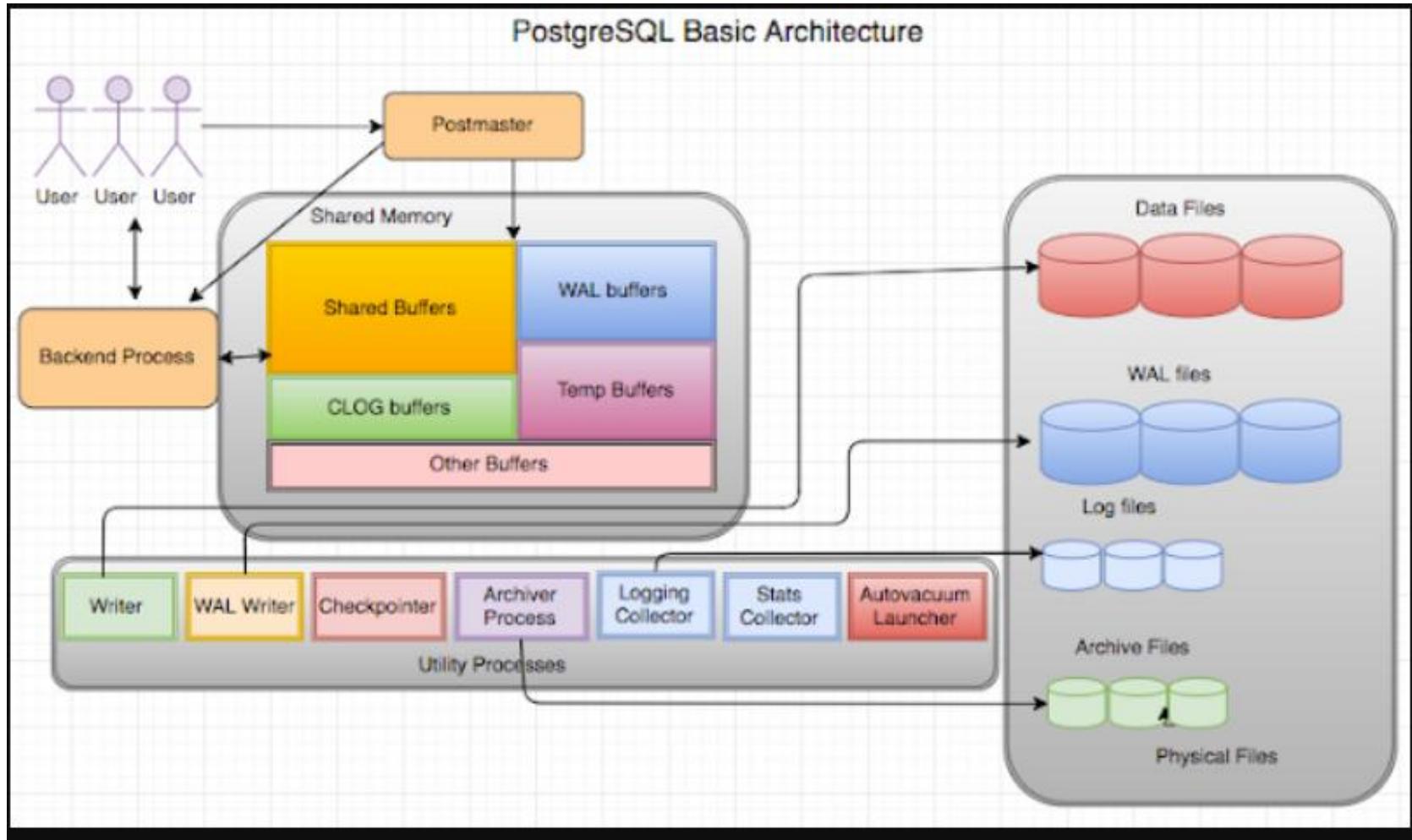
Client side Process

- Whenever we issue a query, or the action made by us (client) is called the client process
- It is front end.
- Front end may be a text application, graphical application or web server page.
- Through TCP/IP clients access the server
- Many users at a time can access the DB
- FORKS – This process makes multiuser access possible. It don't disturb the postgres process

Client side Process

- Client Process refers to the background process that is assigned for every backend user connection.
- Usually, the postmaster process will fork a child process that is dedicated to serving a user connection.

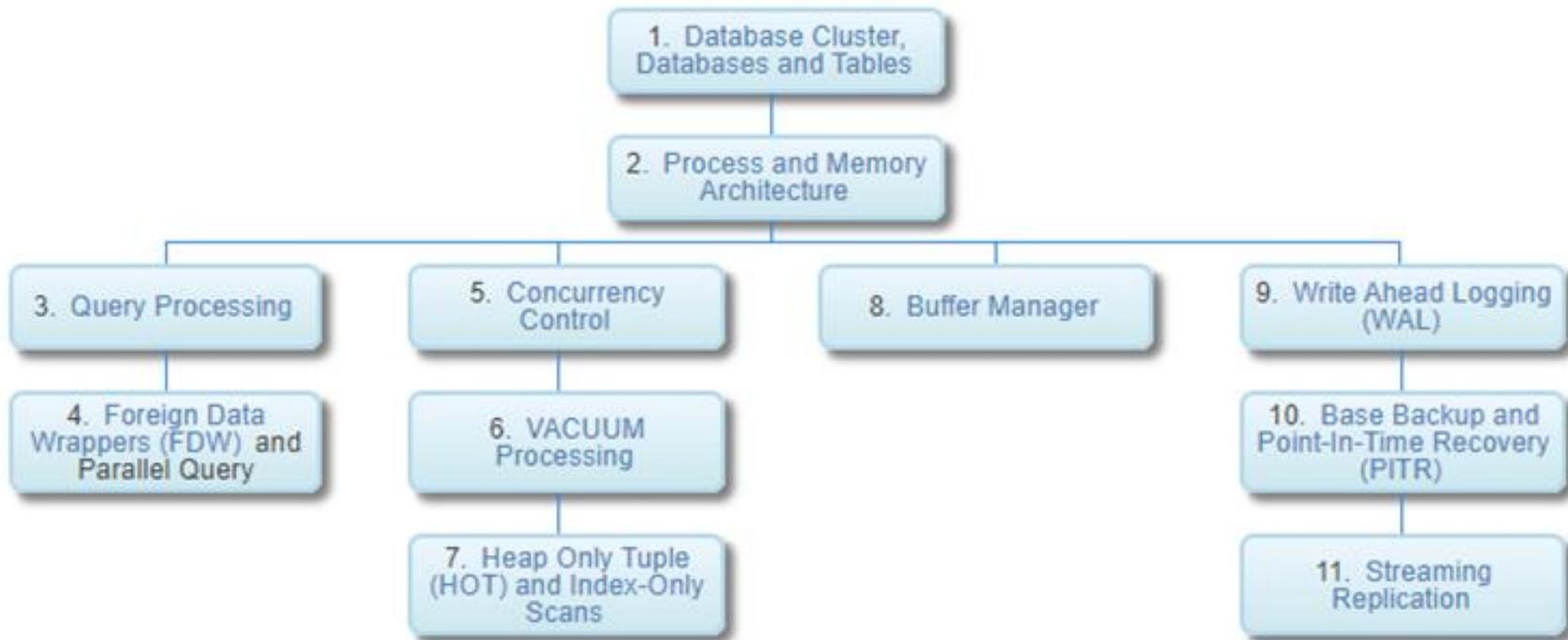
Postgre Architecture



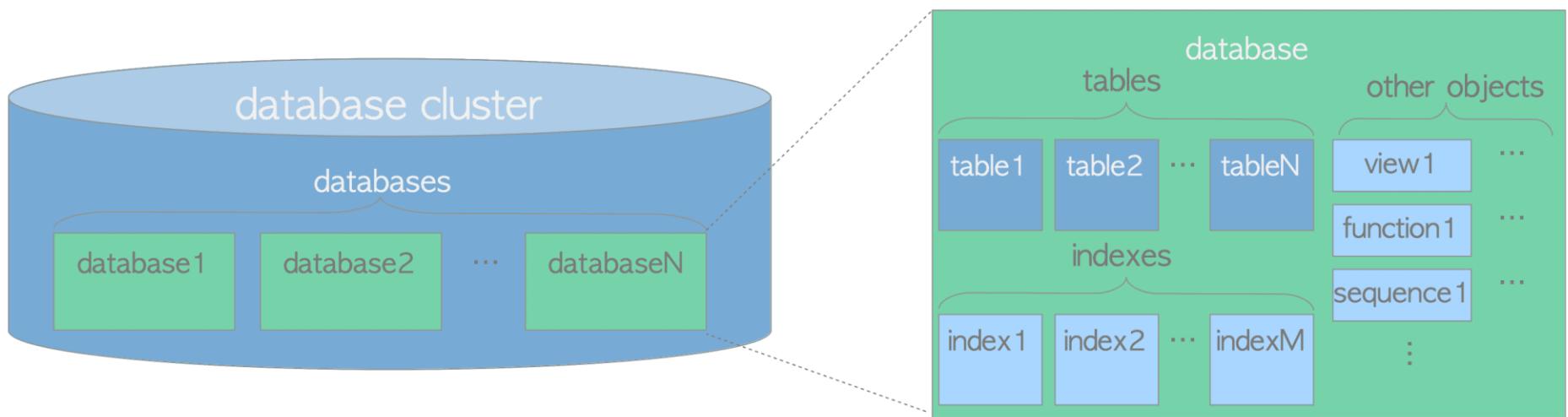
Postgre Architecture

- Processes in PostgreSQL can be divided mainly into 4 types:
 - 1. Postmaster (Daemon) Process
 - 2. Background (Utility) Process
 - 3. Backend Process
 - 4. Client Process

Postgre Architecture Hierarchy



Database Cluster, Databases, and Tables



Database Cluster, Databases, and Tables

- A database cluster is a collection of databases managed by a PostgreSQL server.
- The term ‘database cluster’ in PostgreSQL does not mean ‘a group of database servers’.
- A PostgreSQL server runs on a single host and manages a single database cluster.

Database Cluster, Databases, and Tables

- A database is a collection of database objects.
- In the relational database theory, a database object is a data structure used either to store or to reference data.
- A (heap) table is a typical example of it, and there are many more like an index, a sequence, a view, a function and so on.
- In PostgreSQL, databases themselves are also database objects and are logically separated from each other.
- All other database objects (e.g., tables, indexes, etc) belong to their respective databases.

Database Cluster, Databases, and Tables

- All the database objects in PostgreSQL are internally managed by respective object identifiers (OIDs), which are unsigned 4-byte integers.
- The relations between database objects and the respective OIDs are stored in appropriate system catalogs, depending on the type of objects.
- For example, OIDs of databases and heap tables are stored in `pg_database` and `pg_class` respectively,

Database Cluster, Databases, and Tables

- All the database objects in PostgreSQL are internally managed by respective object identifiers (OIDs), which are unsigned 4-byte integers.
- The relations between database objects and the respective OIDs are stored in appropriate system catalogs, depending on the type of objects.
- For example, OIDs of databases and heap tables are stored in `pg_database` and `pg_class` respectively,

Database Cluster, Databases, and Tables

The screenshot shows the pgAdmin interface for PostgreSQL management. On the left, the sidebar displays the database structure under 'Servers (3)'. The 'PostgreSQL 13' server is expanded, showing 'Databases (3)' which include 'pem', 'postgres', and 'testdb'. Each database has its own set of objects like Casts, Catalogs, Event Triggers, Extensions, Languages, and Schemas.

The main window is titled 'Create - Login/Group Role' and is currently on the 'General' tab. It prompts for a 'Name' (left empty) and 'Comments' (also empty). Below these are two small line charts labeled 'Server sessions' and 'Tuples in'. A large button at the bottom right of this dialog says 'Save'.

A modal dialog is overlaid on the main window, asking 'Save password?'. It contains fields for 'Username' (set to 'admin') and 'Password' (set to '*****'). There are two buttons: 'Save' (highlighted in blue) and 'Never'. To the right of the modal is a performance monitoring panel showing a bar chart for 'Reads' and 'Hits' over time, with values ranging from 0 to 120,000.

At the bottom of the screen, there is a table titled 'Server activity' with columns for 'Sessions', 'Loc', 'PID', 'User', 'Host', 'State', 'Wait event', and 'Blocking PIDs'. It lists two entries:

Sessions	Loc	PID	User	Host	State	Wait event	Blocking PIDs
		3504	pem	agent1	Postgres Enterprise Manager - Agent (SM... 127.0.0.1 2021-03-29 18:26:22 IST	idle	Client: ClientRead
		4316	pem	agent1	Postgres Enterprise Manager - Agent (SN... 127.0.0.1 2021-03-29 18:26:20 IST	idle	Client: ClientRead

Database Cluster, Databases, and Tables

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]: admin
Password for user admin:
psql (13.2)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, compression: off)
Type "help" for help.

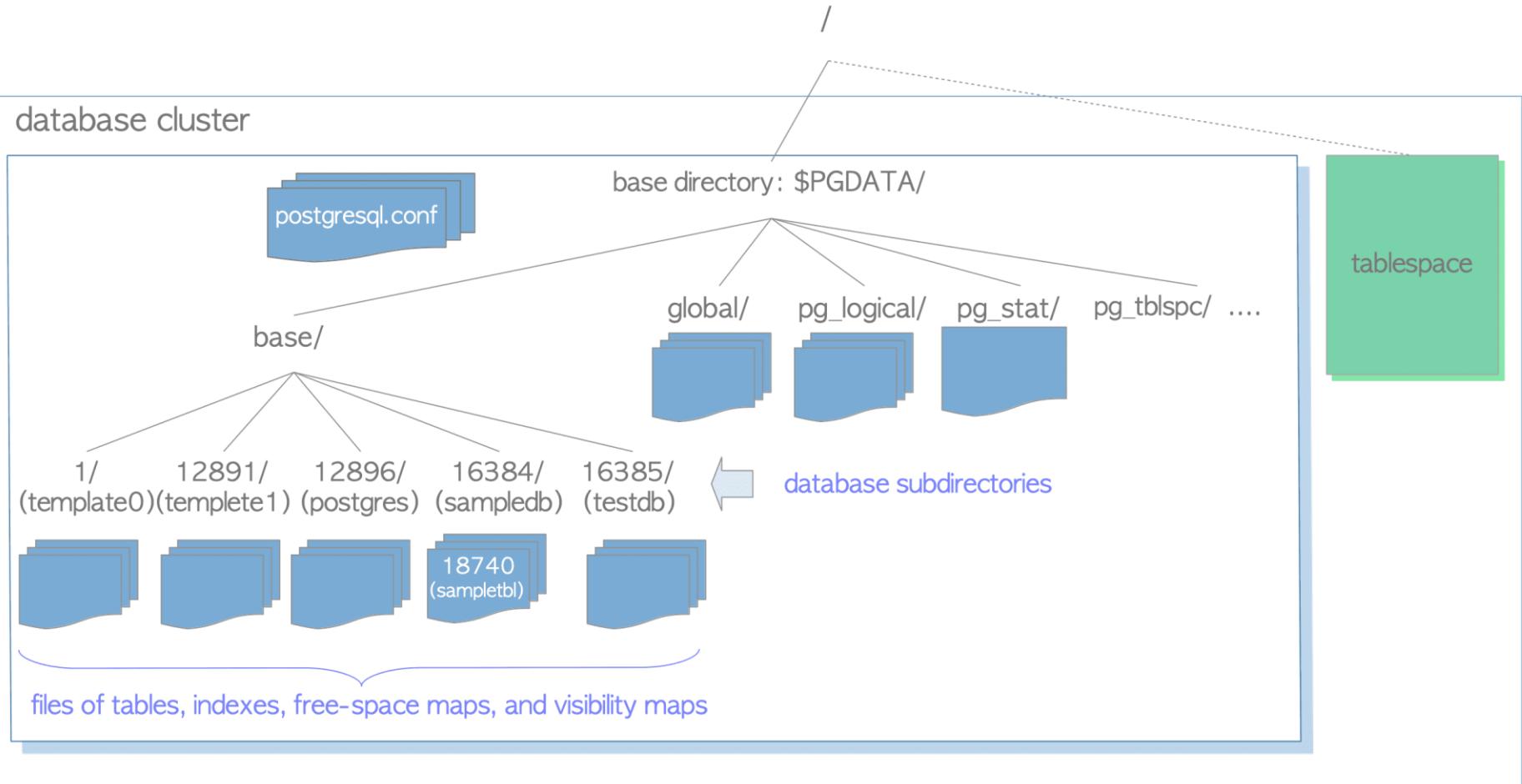
postgres=# SELECT datname, oid FROM pg_database WHERE datname = 'testdb';
 datname | oid
-----+-----
 testdb | 28956
(1 row)

postgres=#
```

Physical Structure of Database Cluster

- A database cluster basically is one directory referred to as base directory, and it contains some subdirectories and lots of files.
- If we execute the initdb utility to initialize a new database cluster, a base directory will be created under the specified directory.
- Path of the base directory is usually set to the environment variable PGDATA.

Physical Structure of Database Cluster



Layout of a Database Cluster

table 1.1: Layout of files and subdirectories under the base directory (From the official document)

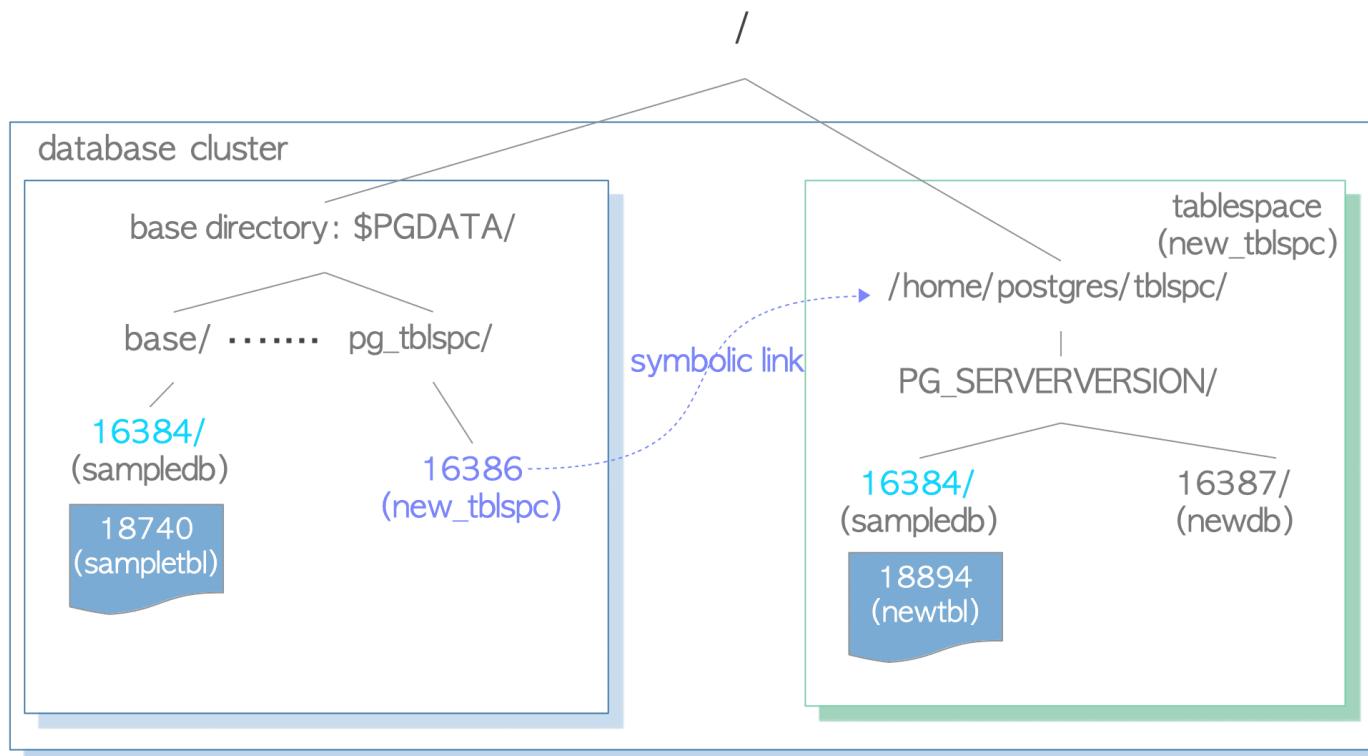
files	description
PG_VERSION	A file containing the major version number of PostgreSQL
pg_hba.conf	A file to control PostgreSQL's client authentication
pg_ident.conf	A file to control PostgreSQL's user name mapping
postgresql.conf	A file to set configuration parameters
postgresql.auto.conf	A file used for storing configuration parameters that are set in ALTER SYSTEM (version 9.4 or later)
postmaster.opts	A file recording the command line options the server was last started with
subdirectories	description
base/	Subdirectory containing per-database subdirectories.
global/	Subdirectory containing cluster-wide tables, such as pg_database and pg_control.
pg_commit_ts/	Subdirectory containing transaction commit timestamp data. Version 9.5 or later.
pg_clog/ (Version 9.6 or earlier)	Subdirectory containing transaction commit state data. It is renamed to <i>pg_xact</i> in Version 10. CLOG will be described in Section 5.4.
pg_dynshmem/	Subdirectory containing files used by the dynamic shared memory subsystem. Version 9.4 or later.
pg_logical/	Subdirectory containing status data for logical decoding. Version 9.4 or later.
pg_multixact/	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify/	Subdirectory containing LISTEN/NOTIFY status data
pg_repslot/	Subdirectory containing replication slot data. Version 9.4 or later.
pg_serial/	Subdirectory containing information about committed serializable transactions (version 9.1 or later)
pg_snapshots/	Subdirectory containing exported snapshots (version 9.2 or later). The PostgreSQL's function pg_export_snapshot creates a snapshot information file in this subdirectory.
pg_stat/	Subdirectory containing permanent files for the statistics subsystem.
pg_stat_tmp/	Subdirectory containing temporary files for the statistics subsystem.

Layout of a Database Cluster

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/ (Version 10 or later)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed from <i>pg_xlog</i> in Version 10.
pg_xact/ (Version 10 or later)	Subdirectory containing transaction commit state data. It is renamed from <i>pg_clog</i> in Version 10. CLOG will be described in Section 5.4.
pg_xlog/ (Version 9.6 or earlier)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed to <i>pg_wal</i> in Version 10.

Tablespaces

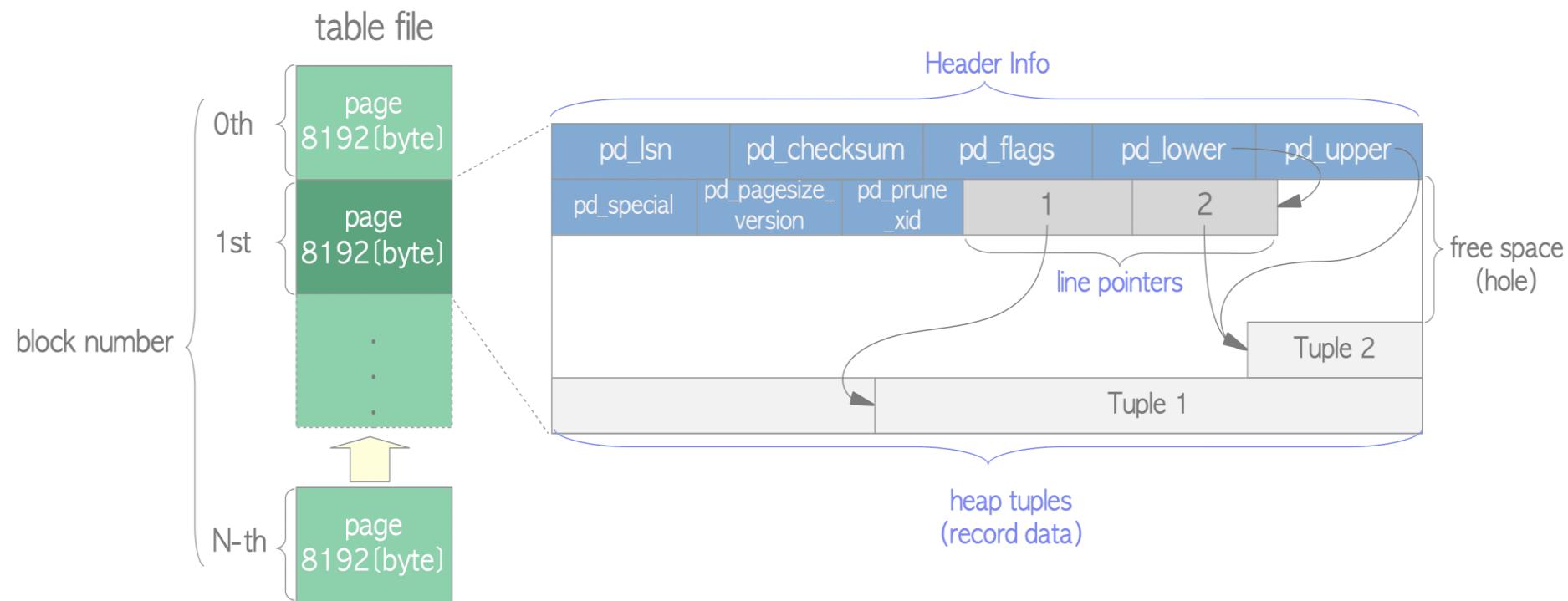
- A tablespace in PostgreSQL is an additional data area outside the base directory.
- This function has been implemented in version 8.0.



Internal Layout of a Heap Table File

- Inside the data file (heap table and index, as well as the free space map and visibility map), it is divided into pages (or blocks) of fixed length, the default is 8192 byte (8 KB).
- Those pages within each file are numbered sequentially from 0, and such numbers are called as block numbers.
- If the file has been filled up, PostgreSQL adds a new empty page to the end of the file to increase the file size.

Internal Layout of a Heap Table File



Internal Layout of a Heap Table File

- heap tuple(s) – A heap tuple is a record data itself. They are stacked in order from the bottom of the page
- line pointer(s) – A line pointer is 4 byte long and holds a pointer to each heap tuple. It is also called an item pointer.
- Line pointers form a simple array, which plays the role of index to the tuples. Each index is numbered sequentially from 1, and called offset number. When a new tuple is added to the page, a new line pointer is also pushed onto the array to point to the new one.

Internal Layout of a Heap Table File

- header data – A header data defined by the structure `PageHeaderData` is allocated in the beginning of the page. It is 24 byte long and contains general information about the page.
- The major variables of the structure are described below.
 - `pd_lsn` – This variable stores the LSN of XLOG record written by the last change of this page. It is an 8-byte unsigned integer, related to the WAL (Write-Ahead Logging) mechanism. The details are described in Chapter 9.
 - `pd_checksum` – This variable stores the checksum value of this page. (Note that this variable is supported in version 9.3 or later; in earlier versions, this part had stored the `timelineld` of the page.)

Internal Layout of a Heap Table File

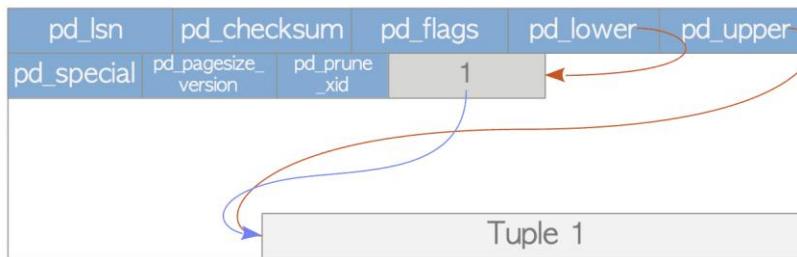
- pd_lower, pd_upper – pd_lower points to the end of line pointers, and pd_upper to the beginning of the newest heap tuple.
- pd_special – This variable is for indexes. In the page within tables, it points to the end of the page. (In the page within indexes, it points to the beginning of special space which is the data area held only by indexes and contains the particular data according to the kind of index types such as B-tree, GiST, GiN, etc.)

Internal Layout of a Heap Table File

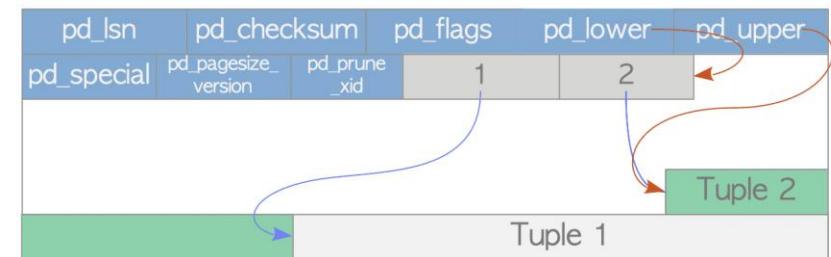
- An empty space between the end of line pointers and the beginning of the newest tuple is referred to as free space or hole.
- To identify a tuple within the table, tuple identifier (TID) is internally used.
- A TID comprises a pair of values: the block number of the page that contains the tuple, and the offset number of the line pointer that points to the tuple.

The Methods of Writing and Reading Tuples

(a) Before insertion of Tuple 2



(b) After insertion of Tuple 2



The Methods of Writing and Reading Tuples

- Writing Heap Tuples
 - Suppose a table composed of one page which contains just one heap tuple.
 - The pd_lower of this page points to the first line pointer, and both the line pointer and the pd_upper point to the first heap tuple.
 - When the second tuple is inserted, it is placed after the first one.
 - The second line pointer is pushed onto the first one, and it points to the second tuple.
 - The pd_lower changes to point to the second line pointer, and the pd_upper to the second heap tuple.

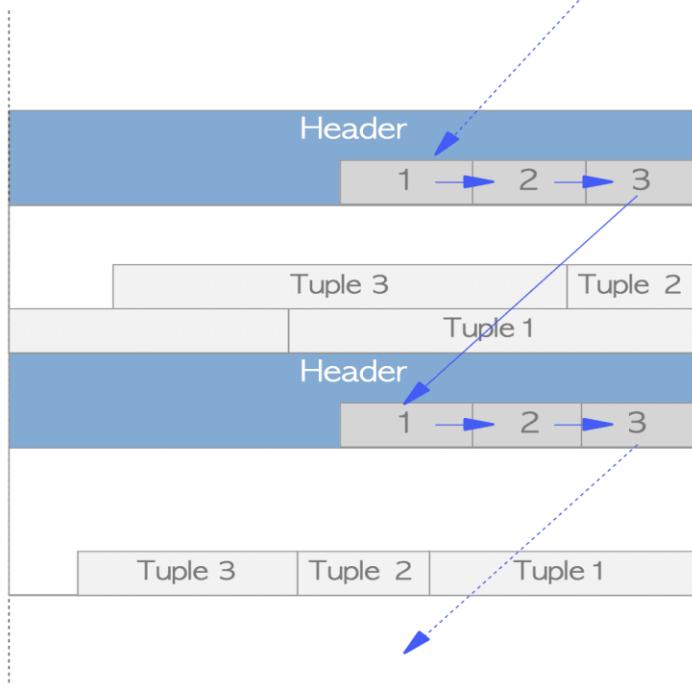
The Methods of Writing and Reading Tuples

- Reading Heap Tuples
 - Two typical access methods, sequential scan and B-tree index scan, are outlined here:
 - Sequential scan – All tuples in all pages are sequentially read by scanning all line pointers in each page.
 - B-tree index scan – An index file contains index tuples, each of which is composed of an index key and a TID pointing to the target heap tuple.
 - If the index tuple with the key that you are looking for has been found, PostgreSQL reads the desired heap tuple using the obtained TID value.
 - It means that the target heap tuple is 2nd tuple in the 7th page within the table, so PostgreSQL can read the desired heap tuple without unnecessary scanning in the pages.

The Methods of Writing and Reading Tuples

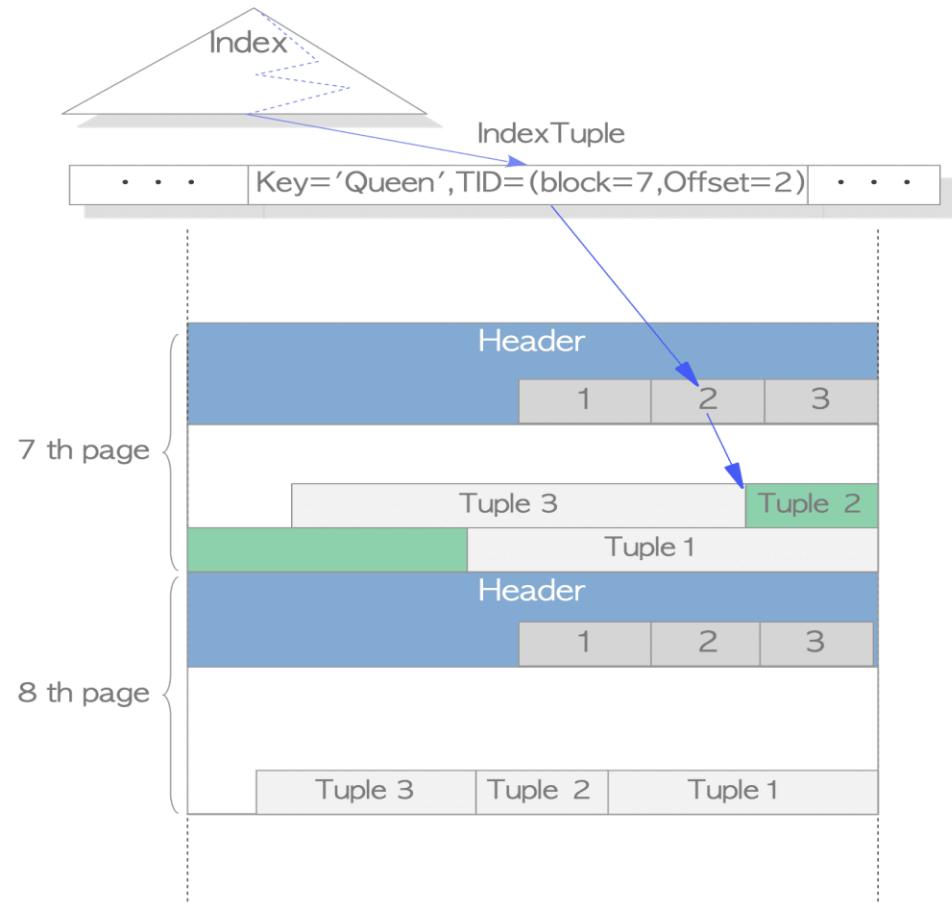
(a) Sequential Scan

SELECT * FROM tbl;



(b) Index Scan

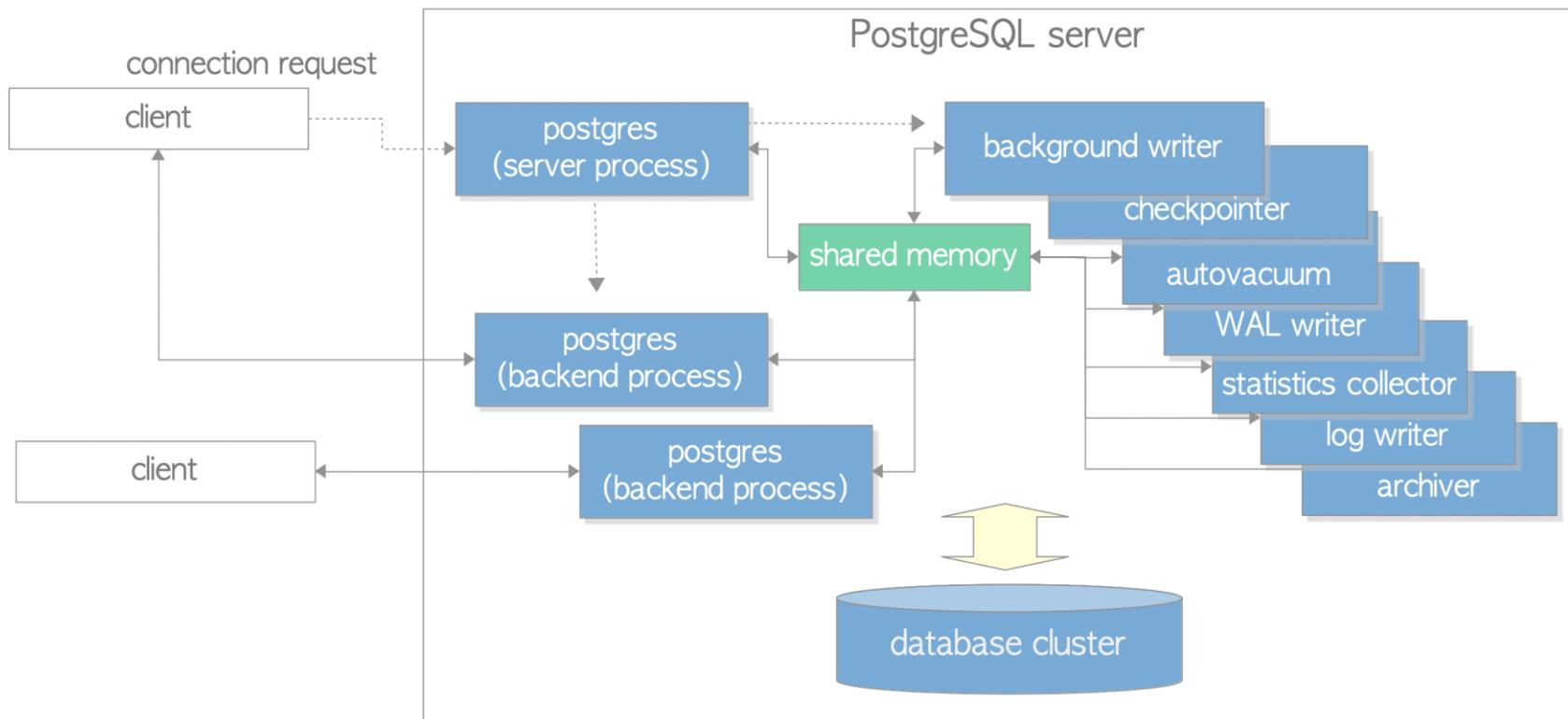
SELECT * FROM tbl WHERE col = 'Queen';



Process and Memory Architecture

- PostgreSQL is a client/server type relational database management system with the multi-process architecture and runs on a single host.
- A collection of multiple processes cooperatively managing one database cluster is usually referred to as a 'PostgreSQL server', and it contains the following types of processes:
 - A `postgres` server process is a parent of all processes related to a database cluster management.
 - Each backend process handles all queries and statements issued by a connected client.
 - Various background processes perform processes of each feature (e.g., `VACUUM` and `CHECKPOINT` processes) for database management.
 - In the replication associated processes, they perform the streaming replication.
 - In the background worker process supported from version 9.3, it can perform any processing implemented by users. As not going into detail here, refer to the official document.

Process and Memory Architecture



Postgres Server Process

- A postgres server process is a parent of all in a PostgreSQL server.
- In the earlier versions, it was called ‘postmaster’.
- By executing the pg_ctl utility with start option, a postgres server process starts up.
- Then, it allocates a shared memory area in memory, starts various background processes, starts replication associated processes and background worker processes if necessary, and waits for connection requests from clients.
- Whenever receiving a connection request from a client, it starts a backend process. (And then, the started backend process handles all queries issued by the connected client.)
- A postgres server process listens to one network port, the default port is 5432. Although more than one PostgreSQL server can be run on the same host, each server should be set to listen to different port number in each other, e.g., 5432, 5433, etc.

Backend Processes

- A backend process, which is also called `postgres`, is started by the `postgres` server process and handles all queries issued by one connected client.
- It communicates with the client by a single TCP connection, and terminates when the client gets disconnected.
- As it is allowed to operate only one database, you have to specify a database you want to use explicitly when connecting to a PostgreSQL server.
- PostgreSQL allows multiple clients to connect simultaneously; the configuration parameter `max_connections` controls the maximum number of the clients (default is 100).
- If many clients such as WEB applications frequently repeat the connection and disconnection with a PostgreSQL server, it increases both costs of establishing connections and of creating backend processes because PostgreSQL has not implemented a native connection pooling feature.
- Such circumstance has a negative effect on the performance of database server. To deal with such a case, a pooling middleware (either `pgbouncer` or `pgpool-II`) is usually used.

Backend Processes

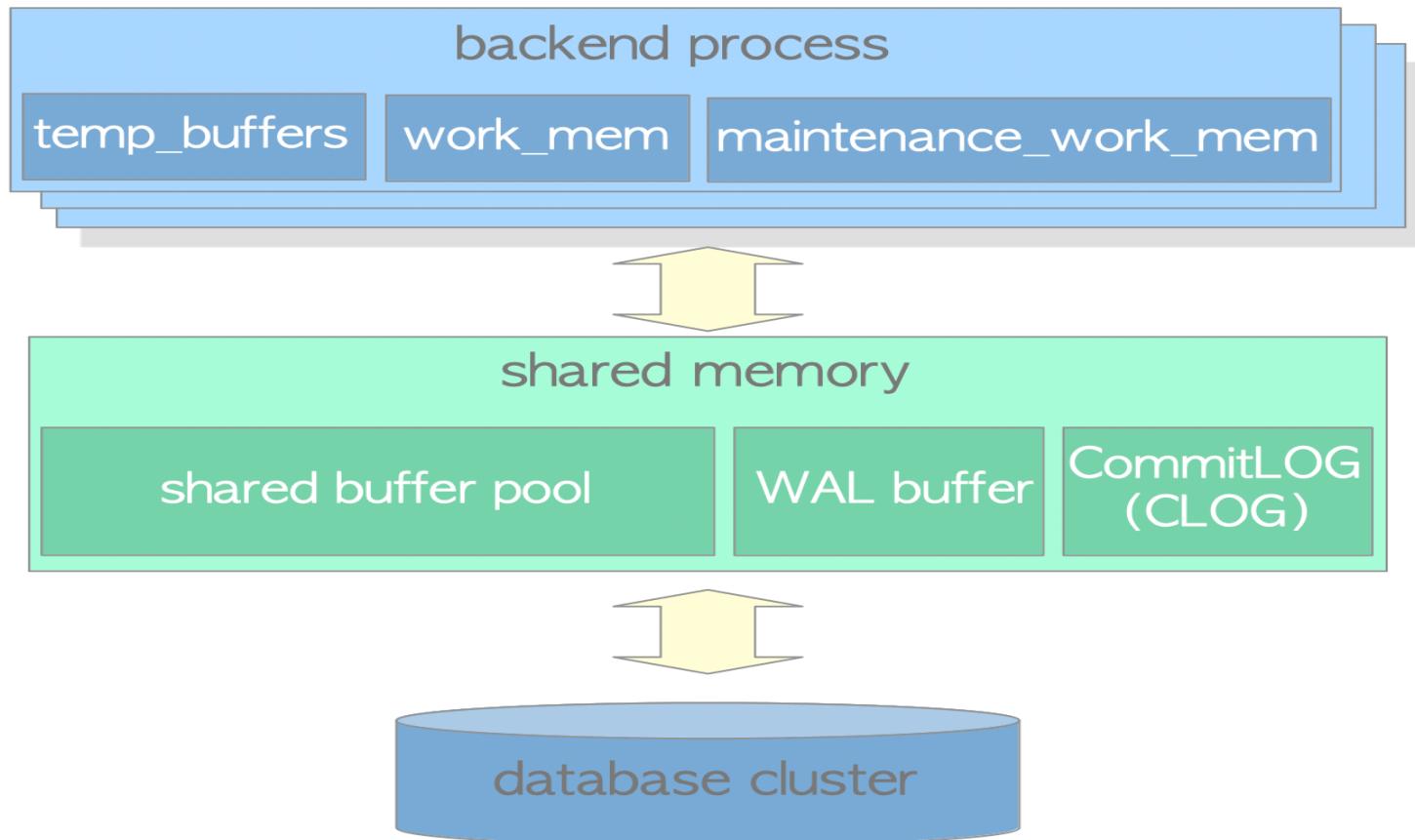
Table 2.1: background processes.

process	description
background writer	In this process, dirty pages on the shared buffer pool are written to a persistent storage (e.g., HDD, SSD) on a regular basis gradually. (In version 9.1 or earlier, it was also responsible for checkpoint process.)
checkpointer	In this process in version 9.2 or later, checkpoint process is performed.
autovacuum launcher	The autovacuum-worker processes are invoked for vacuum process periodically. (More precisely, it requests to create the autovacuum workers to the postgres server.)
WAL writer	This process writes and flushes periodically the WAL data on the WAL buffer to persistent storage.
statistics collector	In this process, statistics information such as for pg_stat_activity and for pg_stat_database, etc. is collected.
logging collector (logger)	This process writes error messages into log files.
archiver	In this process, archiving logging is executed.

Memory Architecture

- Memory architecture in PostgreSQL can be classified into two broad categories:
 - Local memory area – allocated by each backend process for its own use.
 - Shared memory area – used by all processes of a PostgreSQL server.

Memory Architecture



Memory Architecture

- Local Memory Area
 - Each backend process allocates a local memory area for query processing; each area is divided into several sub-areas – whose sizes are either fixed or variable.
- Shared Memory Area
 - A shared memory area is allocated by a PostgreSQL server when it starts up. This area is also divided into several fix sized sub-areas.

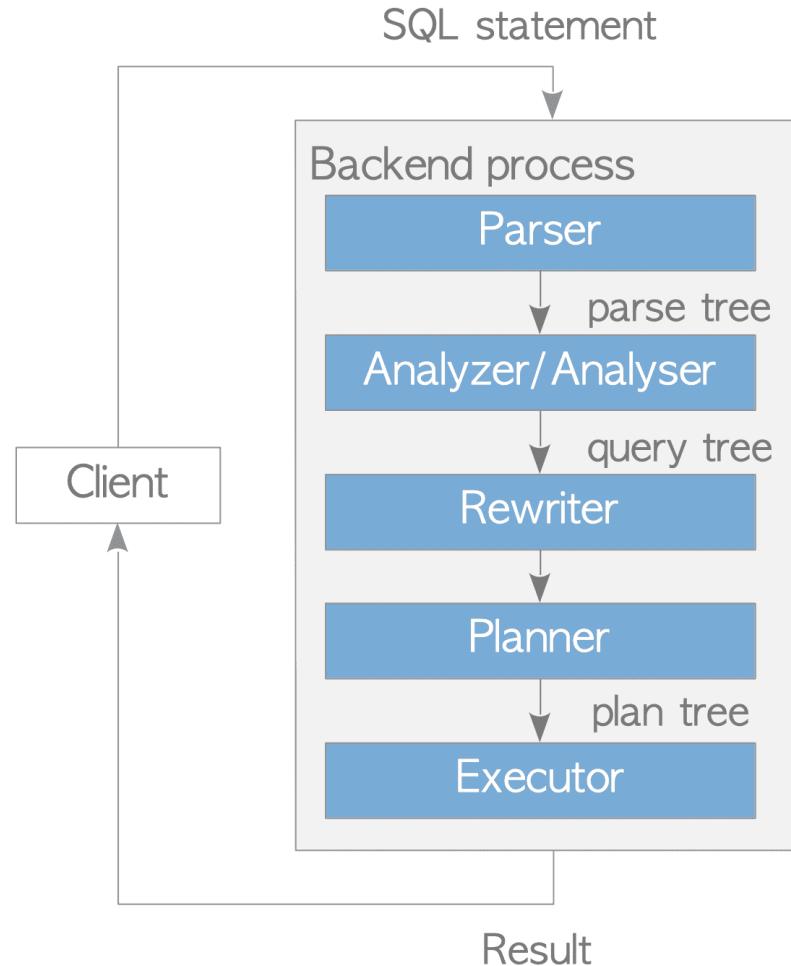
Memory Architecture

sub-area	description
work_mem	Executor uses this area for sorting tuples by ORDER BY and DISTINCT operations, and for joining tables by merge-join and hash-join operations.
maintenance_work_mem	Some kinds of maintenance operations (e.g., VACUUM, REINDEX) use this area.
temp_buffers	Executor uses this area for storing temporary tables.

sub-area	description
shared buffer pool	PostgreSQL loads pages within tables and indexes from a persistent storage to here, and operates them directly.
WAL buffer	To ensure that no data has been lost by server failures, PostgreSQL supports the WAL mechanism. WAL data (also referred to as XLOG records) are transaction log in PostgreSQL; and WAL buffer is a buffering area of the WAL data before writing to a persistent storage.
commit log	Commit Log(CLOG) keeps the states of all transactions (e.g., in_progress,committed,aborted) for Concurrency Control (CC) mechanism.

In addition to them, PostgreSQL allocates several areas as shown below:

Query Processing

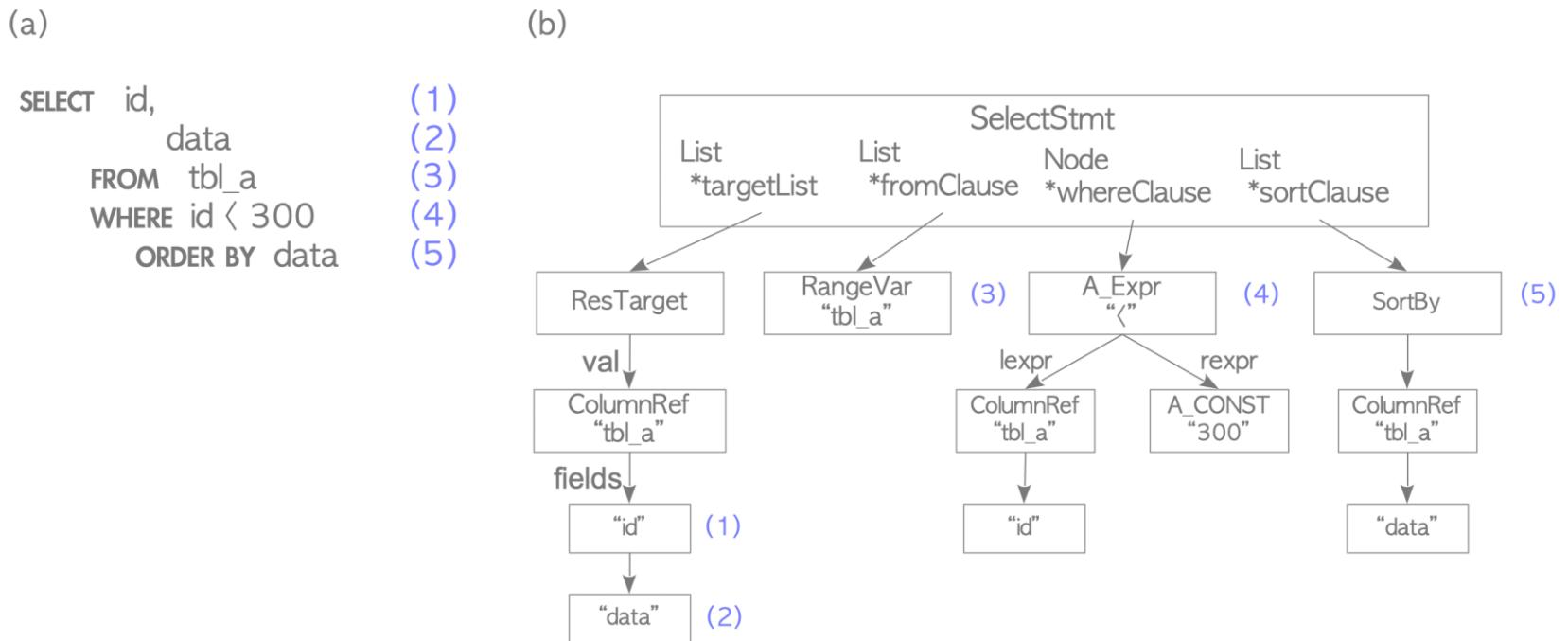


Query Processing

- Parser
 - The parser generates a parse tree from an SQL statement in plain text.
- Analyzer/Analyser
 - The analyzer/analyser carries out a semantic analysis of a parse tree and generates a query tree.
- Rewriter
 - The rewriter transforms a query tree using the rules stored in the rule system if such rules exist.
- Planner
 - The planner generates the plan tree that can most effectively be executed from the query tree.
- Executor
 - The executor executes the query via accessing the tables and indexes in the order that was created by the plan tree.

Query Processing

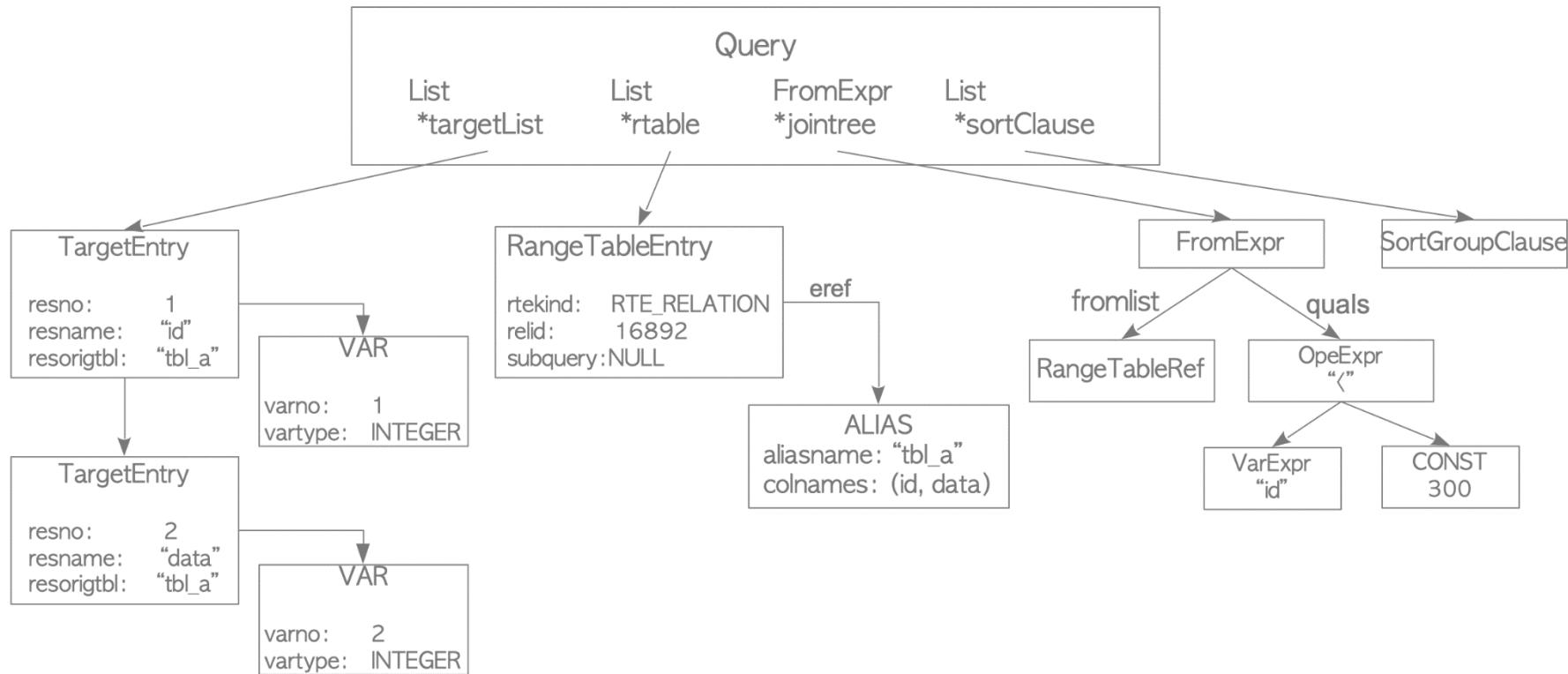
- Parser
 - The parser generates a parse tree that can be read by subsequent subsystems from an SQL statement in plain text.



Query Processing

- Analyzer/Analyser
 - The analyzer/analyser runs a semantic analysis of a parse tree generated by the parser and generates a query tree.
 - The root of a query tree is the Query structure defined in `parsenodes.h`; this structure contains metadata of its corresponding query such as the type of this command (SELECT, INSERT or others) and several leaves; each leaf forms a list or a tree and holds data of the individual particular clause.

Query Processing



Query Processing

- The target list is a list of columns that are the result of this query. In this example, this list is composed of two columns: 'id' and 'data'.
- If the input query tree uses '*' (asterisk), the analyzer/analyser will explicitly replace it to all of the columns.
- The range table is a list of relations that are used in this query.
- In this example, this table holds the information of the table 'tbl_a' such as the oid of this table and the name of this table.
- The join tree stores the FROM clause and the WHERE clauses.
- The sort clause is a list of SortGroupClause.

Architectural Fundamentals

- In database jargon, PostgreSQL uses a client/server model.
- A PostgreSQL session consists of the following cooperating processes (programs):
 - A server process, which manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients.
 - The database server program is called `postgres`.
 - The user's client (frontend) application that wants to perform database operations.
- Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool.
- Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

Architectural Fundamentals

- As is typical of client/server applications, the client and the server can be on different hosts.
- In that case they communicate over a TCP/IP network connection.
- As files can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

Architectural Fundamentals

- The PostgreSQL server can handle multiple concurrent connections from clients.
- To achieve this, it starts (“forks”) a new process for each connection.
- From that point on, the client and the new server process communicate without intervention by the original postgres process.
- Thus, the master server process is always running, waiting for client connections, whereas client and associated server processes come and go. (All of this is of course invisible to the user. We only mention it here for completeness.)

PostgreSQL vs Other RDBMS

Feature	PostgreSQL	MySQL	Oracle DB	SQL Server	SQLite
Type	Open-source ORDBMS	Open-source RDBMS	Commercial RDBMS	Commercial RDBMS	Embedded RDBMS
Developer	PostgreSQL Global Dev.	Oracle Corporation	Oracle Corporation	Microsoft	D. Richard Hipp (Public)
ACID Compliance	<input checked="" type="checkbox"/> Full	<input checked="" type="checkbox"/> (depends on engine)	<input checked="" type="checkbox"/> Full	<input checked="" type="checkbox"/> Full	<input checked="" type="checkbox"/> Full
Licensing	PostgreSQL License	GPL	Commercial (proprietary)	Commercial (proprietary)	Public Domain
Platform Support	Cross-platform	Cross-platform	Cross-platform	Windows, Linux, macOS	Cross-platform
Extensibility	<input checked="" type="checkbox"/> Highly Extensible	<input checked="" type="checkbox"/> Limited	<input checked="" type="checkbox"/> with PL/SQL	<input checked="" type="checkbox"/> with .NET CLR	<input checked="" type="checkbox"/> Limited
Stored Procedures	<input checked="" type="checkbox"/> PL/pgSQL	<input checked="" type="checkbox"/> (SQL/PSM, limited)	<input checked="" type="checkbox"/> PL/SQL	<input checked="" type="checkbox"/> T-SQL	<input checked="" type="checkbox"/> Limited

Postgre vs Other RDBMS

Feature	PostgreSQL	MySQL	Oracle DB	SQL Server	SQLite
MVCC Support	<input checked="" type="checkbox"/> Native MVCC	<input checked="" type="checkbox"/> (InnoDB only)	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> Simulated
JSON & NoSQL Features	<input checked="" type="checkbox"/> Advanced (JSONB, GIN)	<input checked="" type="checkbox"/> Basic JSON support	<input checked="" type="checkbox"/> Basic	<input checked="" type="checkbox"/> Limited	<input checked="" type="checkbox"/> Basic
Replication	<input checked="" type="checkbox"/> Streaming, Logical	<input checked="" type="checkbox"/> Basic (async)	<input checked="" type="checkbox"/> Advanced	<input checked="" type="checkbox"/> Advanced	<input checked="" type="checkbox"/> None
Partitioning	<input checked="" type="checkbox"/> Declarative	<input checked="" type="checkbox"/> (manual or native)	<input checked="" type="checkbox"/> Advanced	<input checked="" type="checkbox"/> Advanced	<input checked="" type="checkbox"/> None
GIS Support	<input checked="" type="checkbox"/> PostGIS	<input checked="" type="checkbox"/> via plugins	<input checked="" type="checkbox"/> Spatial	<input checked="" type="checkbox"/> Spatial	<input checked="" type="checkbox"/> None
Concurrency	<input checked="" type="checkbox"/> Excellent (MVCC)	<input checked="" type="checkbox"/> Good (InnoDB)	<input checked="" type="checkbox"/> Excellent	<input checked="" type="checkbox"/> Good	<input checked="" type="checkbox"/> Poor

PostgreSQL Strengths

- **Advanced SQL Compliance:** Full support for SQL standard, including window functions, CTEs, and more.
- **Extensibility:** You can define custom data types, operators, functions, and index methods.
- **Strong JSON/JSONB:** Can be used for hybrid NoSQL use-cases.
- **Rich Indexing:** Supports B-tree, GiST, GIN, BRIN, Hash, covering a wide range of use-cases.
- **MVCC:** Superior concurrency without locking readers.
- **Open Source & Free for Commercial Use:** No licensing restrictions.

When to Choose What?

Use Case	Recommended DB	Reason
Web applications needing performance + ease	MySQL	Fast, easy, well-supported in LAMP stacks
Enterprise-grade systems with complex logic	Oracle DB	Strong security, clustering, and tooling
Windows-heavy enterprise systems	SQL Server	Good .NET integration, SSIS/SSRS tools
Research, spatial/geospatial apps	PostgreSQL (PostGIS)	Best GIS support, complex query support
Mobile or embedded applications	SQLite	Lightweight, file-based, no server needed
Cloud-native or analytics-heavy systems	PostgreSQL	Modern features, cloud support, and analytics

Security & Role Management

DB	Role Management	Row-Level Security	Encryption
PostgreSQL	<input checked="" type="checkbox"/> Roles, schemas	<input checked="" type="checkbox"/> Built-in	<input checked="" type="checkbox"/> SSL, TDE via extensions
MySQL	<input checked="" type="checkbox"/> Users, privileges	<input checked="" type="checkbox"/> Limited	<input checked="" type="checkbox"/> SSL only
Oracle	<input checked="" type="checkbox"/> Advanced	<input checked="" type="checkbox"/> VPD, Label Sec	<input checked="" type="checkbox"/> TDE, Data Redaction
SQL Server	<input checked="" type="checkbox"/> Windows/SQL Auth	<input checked="" type="checkbox"/> RLS	<input checked="" type="checkbox"/> TDE, Always Encrypted
SQLite	<input checked="" type="checkbox"/> File-based	<input checked="" type="checkbox"/> None	<input checked="" type="checkbox"/> Application-level only

Key Features of PostgreSQL

- **1. ACID Compliance**
 - Guarantees **Atomicity, Consistency, Isolation, and Durability.**
 - Ensures reliable and safe transactions.
- **2. MVCC (Multi-Version Concurrency Control)**
 - Allows **concurrent reads and writes** without locking.
 - Improves performance in high-concurrency environments.

Key Features of PostgreSQL

- **3. Advanced SQL Support**
- Full support for:
 - **Window Functions**
 - **Common Table Expressions (CTEs)**
 - **Subqueries, Joins, and Set operations**
 - **Stored procedures, triggers, and views**

Key Features of PostgreSQL

- **4. Extensibility**
- Add custom:
 - Data types
 - Functions (in PL/pgSQL, Python, Perl, etc.)
 - Operators
 - Indexing methods
 - Procedural languages

Key Features of PostgreSQL

- **5. Rich Data Types**
- Native support for:
 - Numeric, Integer, Boolean, Text, Date/Time
 - Arrays
 - JSON / JSONB (binary-optimized JSON)
 - hstore (key-value pairs)
 - Geometric types
 - UUID, XML, and more

Key Features of PostgreSQL

- **6. Indexing Options**
- Powerful indexing support:
 - **B-tree** (default)
 - **Hash**
 - **GIN** (for full-text search and JSON)
 - **GiST** (for spatial data)
 - **BRIN** (for large sorted datasets)
 - **Expression & Partial indexes**

Key Features of PostgreSQL

- 7. Full Text Search
 - Built-in full-text search capabilities using GIN/GiST indexes.
 - Ranking, highlighting, stemming, and tokenization.
- 8. JSON & NoSQL Support
 - Native JSON and JSONB data types.
 - Querying, indexing, and storing semi-structured data.
 - Acts as a hybrid SQL + NoSQL database.

Key Features of PostgreSQL

- 9. Data Integrity and Constraints
 - Supports:
 - Primary, foreign, and unique keys
 - Check constraints
 - Exclusion constraints (custom conflict rules)
- 10. Partitioning
 - Declarative partitioning on range/list/hash keys.
 - Efficient data management and performance for large tables.

Key Features of PostgreSQL

- **11. Replication & High Availability**
- Built-in:
 - **Streaming replication**
 - **Logical replication**
 - **Hot standby, Point-in-Time Recovery (PITR)**
- Tools: **pgBackRest, Barman, Patroni, PgBouncer**

Key Features of PostgreSQL

- **12. Security Features**
 - Role-based authentication and permissions
 - SSL encryption
 - **Row-Level Security (RLS)**
 - Integration with LDAP, GSSAPI, PAM

Key Features of PostgreSQL

- **13. Stored Procedures & Triggers**
 - Functions written in PL/pgSQL or other languages
 - BEFORE, AFTER, INSTEAD OF triggers
 - Event-driven automation and data integrity enforcement
- **14. Internationalization & Localization**
 - Unicode support
 - Collation and case-sensitivity customization
 - Multilingual data storage

Key Features of PostgreSQL

- **15. Tools and Ecosystem**
 - GUI: pgAdmin, Dbeaver
 - CLI: psql, pg_dump, pg_restore, vacuumdb
 - Extensions: PostGIS, pg_stat_statements, TimescaleDB
- **16. Open Source with Strong Community**
 - PostgreSQL Global Development Group
 - Active mailing lists, regular updates, and transparent roadmap
 - Free for commercial and personal use

Writing Simple Queries

- Creating a Database
 - To create a new database, in this example named mydb, you use the following command:
 - \$ create database mydb;
- Accessing a Database
 - Running the PostgreSQL interactive terminal program, called psql, which allows you to interactively enter, edit, and execute SQL commands.
 - Using an existing graphical frontend tool like pgAdmin or an office suite with ODBC or JDBC support to create and manipulate a database.

Writing Simple Queries

- Accessing a Database
 - \c mydb;

```
mydb=> SELECT version();
                                         version
-----
-----
PostgreSQL 13.2 on x86_64-pc-linux-gnu, compiled by gcc (Debian
4.9.2-10) 4.9.2, 64-bit
(1 row)

mydb=> SELECT current_date;
          date
-----
2016-01-07
(1 row)

mydb=> SELECT 2 + 2;
?column?
-----
        4
(1 row)
```

Writing Simple Queries

The `psql` program has a number of internal commands that are not SQL commands. They begin with the backslash character, “\”. For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

To get out of `psql`, type:

```
mydb=> \q
```

Data Types

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(n)]		fixed-length bit string
bit varying [(n)]	varbit [(n)]	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data (“byte array”)
character [(n)]	char [(n)]	fixed-length character string
character varying [(n)]	varchar [(n)]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer

Data Types

<code>interval [fields]</code> [(p)]		time span
<code>json</code>		textual JSON data
<code>jsonb</code>		binary JSON data, decomposed
<code>line</code>		infinite line on a plane
<code>lseg</code>		line segment on a plane
<code>macaddr</code>		MAC (Media Access Control) address
<code>macaddr8</code>		MAC (Media Access Control) address (EUI-64 format)
<code>money</code>		currency amount
<code>numeric [(p, s)]</code>	<code>decimal</code> [(p, s)]	exact numeric of selectable precision
<code>path</code>		geometric path on a plane
<code>pg_lsn</code>		PostgreSQL Log Sequence Number
<code>pg_snapshot</code>		user-level transaction ID snapshot
<code>point</code>		geometric point on a plane

Data Types

Name	Aliases	Description
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(p)] [without time zone]		time of day (no time zone)
time [(p)] with time zone	timetz	time of day, including time zone
timestamp [(p)] [without time zone]		date and time (no time zone)
timestamp [(p)] with time zone	timestamptz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot (deprecated; see pg_snapshot)
uuid		universally unique identifier
xml		XML data

Numeric Types

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	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
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Monetary Types

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

Binary Input Type

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

Character Types

Name	Description
character varying (<i>n</i>) , varchar (<i>n</i>)	variable-length with limit
character (<i>n</i>) , char (<i>n</i>)	fixed-length, blank padded
text	variable unlimited length

Boolean Type

Name	Storage Size	Description
boolean	1 byte	state of true or false

Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(p)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond
timestamp [(p)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond
date	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond
time [(p)] with time zone	12 bytes	time of day (no date), with time zone	00:00:00+1559	24:00:00-1559	1 microsecond
interval [fields] [(p)]	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond

Date/Time Types

Example	Description
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
January 8, 1999	unambiguous in any <code>datestyle</code> input mode
1/8/1999	January 8 in MDY mode; August 1 in DMY mode
1/18/1999	January 18 in MDY mode; rejected in other modes
01/02/03	January 2, 2003 in MDY mode; February 1, 2003 in DMY mode; February 3, 2001 in YMD mode
1999-Jan-08	January 8 in any mode

Time Input

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	same as 04:05; AM does not affect value
04:05 PM	same as 16:05; input hour must be <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	time zone specified by abbreviation
2003-04-12 04:05:06 America/New_York	time zone specified by full name

Oracle Data Types vs Postgre Data Types

Oracle type	Possible PostgreSQL types
CHAR	char, varchar, text
NCHAR	char, varchar, text
VARCHAR	char, varchar, text
VARCHAR2	char, varchar, text, json
NVARCHAR2	char, varchar, text
CLOB	char, varchar, text, json
LONG	char, varchar, text
RAW	uuid, bytea
BLOB	bytea
BFILE	bytea (read-only)
LONG RAW	bytea
NUMBER	numeric, float4, float8, char, varchar, text
NUMBER(n,m) with m<=0	numeric, float4, float8, int2, int4, int8,
	boolean, char, varchar, text
FLOAT	numeric, float4, float8, char, varchar, text

Oracle Data Types vs Postgre Data Types

FLOAT	numeric, float4, float8, char, varchar, text
BINARY_FLOAT	numeric, float4, float8, char, varchar, text
BINARY_DOUBLE	numeric, float4, float8, char, varchar, text
DATE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH TIME ZONE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH	date, timestamp, timestamptz, char, varchar, text
LOCAL TIME ZONE	
INTERVAL YEAR TO MONTH	interval, char, varchar, text
INTERVAL DAY TO SECOND	interval, char, varchar, text
MDSYS.SDO_GEOMETRY	geometry (see "PostGIS support" below)

Enumerated Type

Declaration of Enumerated Types

Enum types are created using the CREATE TYPE command, for example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Once created, the enum type can be used in table and function definitions much like any other type:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
   name | current_mood
-----+-----
 Moe   | happy
(1 row)
```

Geometric Types

Name	Storage Size	Description	Representation
point	16 bytes	Point on a plane	(x,y)
line	32 bytes	Infinite line	{A,B,C}
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))
box	32 bytes	Rectangular box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	Circle	<(x,y),r> (center point and radius)

Geometric Types

```
CREATE EXTENSION postgis;
```

```
CREATE TABLE geometries (name varchar, geom geometry);
```

```
INSERT INTO geometries VALUES
```

```
('Point', 'POINT(0 0)'),  
('Linestring', 'LINESTRING(0 0, 1 1, 2 1, 2 2)'),  
('Polygon', 'POLYGON((0 0, 1 0, 1 1, 0 1, 0 0)))'),  
('PolygonWithHole', 'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, 1 2, 2 2, 2 1, 1 1)))'),  
('Collection', 'GEOMETRYCOLLECTION(POINT(2 0),POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))))');
```

```
SELECT name, ST_AsText(geom) FROM geometries;
```

Network Address Types

Name	Storage Size	Description
cidr	7 or 19 bytes	IPv4 and IPv6 networks
inet	7 or 19 bytes	IPv4 and IPv6 hosts and networks
macaddr	6 bytes	MAC addresses
macaddr8	8 bytes	MAC addresses (EUI-64 format)

Network Address Types

```
CREATE TABLE inet_test (  
    address INET  
)
```

```
INSERT INTO inet_test (address) VALUES  
('198.24.10.0/24');
```

```
INSERT INTO inet_test (address) VALUES  
('198.24.10.0');
```

```
INSERT INTO inet_test (address) VALUES ('198.10/8');
```

cidr Type Input

cidr Input	cidr Output	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba/64
2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba: 2e0:81ff:fe22:d1f1/128
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

Bit String Types

- Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks.
- There are two SQL bit types: bit(n) and bit varying(n), where n is a positive integer.
- bit type data must match the length n exactly; it is an error to attempt to store shorter or longer bit strings.
- bit varying data is of variable length up to the maximum length n; longer strings will be rejected.
- Writing bit without a length is equivalent to bit(1), while bit varying without a length specification means unlimited length.

Bit String Types

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR: bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
```

UUID Type

- The data type `uuid` stores Universally Unique Identifiers (UUID) as defined by RFC 4122, ISO/IEC 9834-8:2005, and related standards.
- (This identifier is a 128-bit quantity that is generated by an algorithm chosen to make it very unlikely that the same identifier will be generated by anyone else in the known universe using the same algorithm.
- Therefore, for distributed systems, these identifiers provide a better uniqueness guarantee than sequence generators, which are only unique within a single database.

UUID Type

- PostgreSQL also accepts the following alternative forms for input: use of upper-case digits, the standard format surrounded by braces, omitting some or all hyphens, adding a hyphen after any group of four digits.
- Examples are:
- A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
- {a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
- a0eebc999c0b4ef8bb6d6bb9bd380a11
- a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
- {a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}

UUID Type

```
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
SELECT uuid_generate_v1();
SELECT uuid_generate_v4();
CREATE TABLE contacts (
    contact_id uuid DEFAULT uuid_generate_v4 (),
    first_name VARCHAR NOT NULL,
    last_name VARCHAR NOT NULL,
    email VARCHAR NOT NULL,
    phone VARCHAR,
    PRIMARY KEY (contact_id)
);
```

UUID Type

```
INSERT INTO contacts (
    first_name,
    last_name,
    email,
    phone
)
VALUES
(
    'John',
    'Smith',
    'john.smith@example.com',
    '408-237-2345'
),
```

UUID Type

```
(  
    'Jane',  
    'Smith',  
    'jane.smith@example.com',  
    '408-237-2344'  
,  
(  
    'Alex',  
    'Smith',  
    'alex.smith@example.com',  
    '408-237-2343'  
);
```

XML Type

- The xml data type can be used to store XML data.
- Its advantage over storing XML data in a text field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it;
- Use of this data type requires the installation to have been built with configure --with-libxml.

XML Type

- The `xml` type can store well-formed “documents”, as defined by the XML standard, as well as “content” fragments, which are defined by reference to the more permissive “document node”¹ of the XQuery and XPath data model.
- Roughly, this means that content fragments can have more than one top-level element or character node.
- The expression `xmlvalue IS DOCUMENT` can be used to evaluate whether a particular `xml` value is a full document or only a content fragment.

XML Type

To produce a value of type `xml` from character data, use the function `xmlparse`:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Examples:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

While this is the only way to convert character strings into XML values according to the SQL standard, the PostgreSQL-specific syntaxes:

```
xml '<foo>bar</foo>
'<foo>bar</foo>' ::xml
```

can also be used.

Xml Type

The `xml` type does not validate input values against a document type declaration (DTD), even when the input value specifies a DTD. There is also currently no built-in support for validating against other XML schema languages such as XML Schema.

The inverse operation, producing a character string value from `xml`, uses the function `xmlserialize`:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

`type` can be `character`, `character varying`, or `text` (or an alias for one of those). Again, according to the SQL standard, this is the only way to convert between type `xml` and character types, but PostgreSQL also allows you to simply cast the value.

When a character string value is cast to or from type `xml` without going through `XMLEPARSE` or `XMLSERIALIZE`, respectively, the choice of `DOCUMENT` versus `CONTENT` is determined by the “`XML option`” session configuration parameter, which can be set using the standard command:

```
SET XML OPTION { DOCUMENT | CONTENT } ;
```

or the more PostgreSQL-like syntax

```
SET xmloption TO { DOCUMENT | CONTENT } ;
```

The default is `CONTENT`, so all forms of XML data are allowed.

JSON Types

- JSON data types are for storing JSON (JavaScript Object Notation) data, as specified in RFC 71592.
- Such data can also be stored as text, but the JSON data types have the advantage of enforcing that each stored value is valid according to the JSON rules.
- There are also assorted JSON-specific functions and operators available for data stored in these data types;
- PostgreSQL offers two types for storing JSON data: json and jsonb.

JSON Types

- The json and jsonb data types accept almost identical sets of values as input.
- The major practical difference is one of efficiency.
- The json data type stores an exact copy of the input text, which processing functions must reparse on each execution; while jsonb data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed.
- jsonb also supports indexing, which can be a significant advantage.

JSON Types

- Because the json type stores an exact copy of the input text, it will preserve semantically-insignificant white space between tokens, as well as the order of keys within JSON objects.
- Also, if a JSON object within the value contains the same key more than once, all the key/value pairs are kept.
- By contrast, jsonb does not preserve white space, does not preserve the order of object keys, and does not keep duplicate object keys.
- If duplicate keys are specified in the input, only the last value is kept.
- In general, most applications should prefer to store JSON data as jsonb, unless there are quite specialized needs, such as legacy assumptions about ordering of object keys.

JSON Types

JSON primitive type	PostgreSQL type	Notes
string	text	\u0000 is disallowed, as are Unicode escapes representing characters not available in the database encoding
number	numeric	NaN and infinity values are disallowed
boolean	boolean	Only lowercase true and false spellings are accepted
null	(none)	SQL NULL is a different concept

Arrays

```
CREATE TABLE sal_emp (
    name          text,
    pay_by_quarter integer [],
    schedule      text [] []
) ;

INSERT INTO sal_emp
VALUES ('Bill',
'[{10000, 10000, 10000, 10000}]',
'{{{"meeting", "lunch"}, {"training", "presentation"}}}');

INSERT INTO sal_emp
VALUES ('Carol',
'[{20000, 25000, 25000, 25000}]',
'{{{"breakfast", "consulting"}, {"meeting", "lunch"}}}');
```

Sequence

- CREATE [TEMPORARY | TEMP] SEQUENCE [IF NOT EXISTS] name
- [AS data_type]
- [INCREMENT [BY] increment]
- [MINVALUE minvalue | NO MINVALUE][MAXVALUE maxvalue | NO MAXVALUE]
- [START [WITH] start] [CACHE cache] [[NO] CYCLE]
- [OWNED BY { table_name.column_name | NONE }]

Sequence Parameters

- TEMPORARY or TEMP — PostgreSQL can create a temporary sequence within a session. Once the session ends, the sequence is automatically dropped.
- IF NOT EXISTS — Creates a sequence even if a sequence with an identical name already exists. Replaces the existing sequence.
- AS — A new option in PostgreSQL 10. It is for specifying the data type of the sequence. The available options are smallint, integer, and bigint (default). This also determines the maximum and minimum values.
- INCREMENT BY — An optional parameter with a default value of 1. Positive values generate sequence values in ascending order. Negative values generate sequence values in descending sequence.

Sequence Parameters

- START WITH — The same as Oracle. This is an optional parameter having a default of 1. It uses the MINVALUE for ascending sequences and the MAXVALUE for descending sequences.
- MAXVALUE | NO MAXVALUE — Defaults are between 263 for ascending sequences and -1 for descending sequences.
- MINVALUE | NO MINVALUE — Defaults are between 1 for ascending sequences and -263 for descending sequences.
- CYCLE | NO CYCLE — If the sequence value reaches MAXVALUE or MINVALUE, the CYCLE parameter instructs the sequence to return to the initial value (MINVALUE or MAXVALUE). The default is NO CYCLE.

Sequence Parameters

- CACHE — Note that in PostgreSQL, the NOCACHE isn't supported. By default, when not specifying the CACHE parameter, no sequence values will be pre-cached into memory, which is equivalent to the Oracle NOCACHE parameter. The minimum value is 1.
- OWNED BY | OWNBY NON — Specifies that the sequence object is to be associated with a specific column in a table, which isn't supported by Oracle. When dropping this type of sequence, an error will be returned because of the sequence/table association.

Arrays

The result of the previous two inserts looks like this:

```
SELECT * FROM sal_emp;
 name |      pay_by_quarter      |          schedule
-----+-----+
+-----+
 Bill | {10000,10000,10000,10000} | {{meeting,lunch},
{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},
{meeting,lunch}}
(2 rows)
```

Accessing Arrays

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <>  
pay_by_quarter[2];
```

```
name  
-----  
Carol  
(1 row)
```

```
SELECT pay_by_quarter[3] FROM sal_emp;
```

```
pay_by_quarter  
-----  
10000  
25000  
(2 rows)
```

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';
```

```
schedule  
-----  
{ {meeting}, {training} }  
(1 row)
```

Accessing Arrays

```
SELECT schedule[1:2] [2] FROM sal_emp WHERE name = 'Bill';
```

It is possible to omit the *lower-bound* and/or *upper-bound* of a slice specifier; the missing bound is replaced by the lower or upper limit of the array's subscripts. For example:

```
SELECT schedule[:2] [2:] FROM sal_emp WHERE name = 'Bill';
```

```
schedule
-----
{{lunch},{presentation}}
(1 row)
```

```
SELECT schedule[:] [1:1] FROM sal_emp WHERE name = 'Bill';
```

```
schedule
-----
{{meeting},{training}}
(1 row)
```

Accessing Arrays

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';
```

```
array_dims
```

```
-----
```

```
[1:2] [1:2]
```

```
(1 row)
```

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';
```

```
array_length
```

```
-----
```

```
2
```

```
(1 row)
```

cardinality returns the total number of elements in an array across all dimensions. It is effectively the number of rows a call to unnest would yield:

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';
```

```
cardinality
```

```
-----
```

```
4
```

```
(1 row)
```

Modifying Arrays

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'  
WHERE name = 'Carol';
```

or using the ARRAY expression syntax:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]  
WHERE name = 'Carol';
```

An array can also be updated at a single element:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000  
WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'  
WHERE name = 'Carol';
```

Modifying Arrays

```
SELECT ARRAY [1,2] || ARRAY [3,4] ;  
?column?  
-----  
{1,2,3,4}  
(1 row)
```

```
SELECT ARRAY [5,6] || ARRAY [[1,2],[3,4]] ;  
?column?  
-----  
{ {5,6}, {1,2}, {3,4} }  
(1 row)
```

```
SELECT array_dims(ARRAY [1,2] || ARRAY [3,4,5]) ;  
array_dims  
-----  
[1:5]  
(1 row)
```

```
SELECT array_dims(ARRAY [[1,2],[3,4]] || ARRAY [[5,6],[7,8],[9,0]]);  
array_dims  
-----  
[1:5] [1:2]  
(1 row)
```

Modifying Arrays

```
SELECT array_prepend(1, ARRAY[2,3]) ;
array_prepend
-----
{1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3) ;
array_append
-----
{1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]) ;
array_cat
-----
{1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]) ;
array_cat
-----
{{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]) ;
array_cat
-----
{{5,6},{1,2},{3,4}}
```

Searching in Arrays

To search for a value in an array, each value must be checked. This can be done manually, if you know the size of the array. For example:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR  
                      pay_by_quarter[2] = 10000 OR  
                      pay_by_quarter[3] = 10000 OR  
                      pay_by_quarter[4] = 10000;
```

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. An alternative method is described in Section 9.24. The above query could be replaced by:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

In addition, you can find rows where the array has all values equal to 10000 with:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Searching in Arrays

```
SELECT
array_position(ARRAY['sun', 'mon', 'tue', 'wed', 'thu', 'fri', 'sat'],
'mon') ;
```

```
array_position
```

```
-----  
2
```

```
(1 row)
```

```
SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1) ;
```

```
array_positions
```

```
-----  
{1,4,8}
```

```
(1 row)
```

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.
- PostgreSQL allows composite types to be used in many of the same ways that simple types can be used.
- For example, a column of a table can be declared to be of a composite type.

Declaration of Composite Types

```
CREATE TYPE complex AS (
    r          double precision,
    i          double precision
) ;

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

Constructing Composite Values

- To write a composite value as a literal constant, enclose the field values within parentheses and separate them by commas.
- We can put double quotes around any field value, and must do so if it contains commas or parentheses.
- Thus, the general format of a composite constant is the following:
- '(val1 , val2 , ...)'

Constructing Composite Values

An example is:

```
'("fuzzy dice",42,1.99)'
```

which would be a valid value of the `inventory_item` type defined above. To make a field be `NULL`, write no characters at all in its position in the list. For example, this constant specifies a `NULL` third field:

```
'("fuzzy dice",42,)'
```

If you want an empty string rather than `NULL`, write double quotes:

```
'("",42,)'
```

Here the first field is a non-`NULL` empty string, the third is `NULL`.

Accessing Composite Types

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

This will not work since the name `item` is taken to be a table name, not a column name of `on_hand`, per SQL syntax rules. You must write it like this:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

or if you need to use the table name as well (for instance in a multitable query), like this:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price  
> 9.99;
```

Built in Range Types

- int4range — Range of integer
- int8range — Range of bigint
- numrange — Range of numeric
- tsrange — Range of timestamp without time zone
- tstzrange — Range of timestamp with time zone
- daterange — Range of date

Built in Range Types

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30]');

-- Containment
SELECT int4range(10, 20) @> 3;

-- Overlaps
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));
```

PostgreSQL Full Text Search Entities

- Document: A set of data on which your full-text search will take place is called Document.
- In PostgreSQL, you can create a document using either a single column or multiple columns or even using multiple tables.
- Lexemes: The PostgreSQL document during processing is parsed into multiple tokens, which consist of words, sentences, etc. of the text present in the document.
- These tokens are modified to form more meaningful entities called lexemes.

PostgreSQL Full Text Search Entities

- Dictionaries: The conversion of Tokens into Lexemes is done using dictionaries.
- PostgreSQL offers both options of either using the built-in dictionaries or creating your own dictionaries.
- These dictionaries are responsible to determine which stop words are not important and whether differently derived words have the same base language.

to_tsvector

- To create a vector for a sentence, you can use the `to_tsvector` function as follows:
- `SELECT to_tsvector('The quick brown fox jumped over the lazy dog.');`
- `to_tsvector -----'brown':3 'dog':9 'fox':4 'jump':5 'lazi':8 'quick':2`

to_tsquery

- This function accepts a list of words that would be checked against the normalized vector created with the `to_tsvector()` function.
- You can use the following Postgres functions to convert text fields to tsquery values such as `to_tsquery`, `plainto_tsquery`, and `phraseto_tsquery`.
- `SELECT to_tsquery('The quick brown fox jumped over the lazy dog')`

Searching

- After creating tsvector from your text data and tsquery from your search terms, you can further perform a full-text search using the @@ operator.
- @@ operator checks if tsquery matches tsvector.
- For instance, if the word to be queried is “fox” for the above-mentioned example, then:
- `SELECT to_tsvector('The quick brown fox jumped over the lazy dog') @@ to_tsquery('fox');`
- Refer courses table in queries.txt

The SQL Language

- Creating a New Table

```
CREATE TABLE weather (
  city varchar(80),
  temp_lo int, -- low temperature
  temp_hi int, -- high temperature
  prcp real, -- precipitation
  date date
);
```

Populating a Table With Rows

- The INSERT statement is used to populate a table with rows:
- `INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');`
- Note that all data types use rather obvious input formats.
- Constants that are not simple numeric values usually must be surrounded by single quotes ('), as in the example

What is Normalization?

- **Normalization** is the process of organizing data in a database to:
 - Eliminate **redundant data** (repetition).
 - Ensure **data dependencies** make sense.
 - Improve **data integrity**.

Goals of Normalization

- Minimize duplication of data.
- Protect the database against logical or structural anomalies.
- Make database maintenance easier and more efficient.

Normal Forms (NF)

- Normalization is done through a series of **normal forms**, each building on the previous:
-  **1. First Normal Form (1NF)**
- **Atomic values** only (no repeating groups or arrays).
- Each column should contain **unique and indivisible** values.

Normal Forms (NF)

- Normalization is done through a series of **normal forms**, each building on the previous:
-  **1. First Normal Form (1NF)**
- **Atomic values** only (no repeating groups or arrays).
- Each column should contain **unique and indivisible** values.

First Normal Form

Student	Subjects
John	Math, Science

Student	Subject
John	Math
John	Science

2. Second Normal Form (2NF)

- 1NF + every non-key attribute is fully dependent on the entire primary key (no partial dependency).
- Applies only to tables with composite keys.

Violation Example:

OrderID	ProductID	ProductName
---------	-----------	-------------



Split into:

- Orders (OrderID, ProductID)
- Products (ProductID, ProductName)

Third Normal Form (3NF)

- 2NF + no transitive dependencies (non-key attribute depends on another non-key attribute).

Violation Example:

EmployeeID	DepartmentID	DepartmentName
------------	--------------	----------------

 Fix:

- Employee (EmployeeID, DepartmentID)
- Department (DepartmentID, DepartmentName)

Boyce-Codd Normal Form (BCNF)

- BCNF (Boyce-Codd Normal Form) is a stronger version of the Third Normal Form (3NF).
- **Rule:**
A relation is in **BCNF if and only if every determinant is a candidate key.**
- A **determinant** is any attribute (or set of attributes) on which some other attribute is **fully functionally dependent**.
- A **candidate key** is a minimal set of attributes that can uniquely identify a tuple.

Example That Is in 3NF but Not in BCNF

Course	Instructor	Room
DBMS	Alice	R1
OS	Bob	R2
DBMS	Alice	R1

Functional Dependencies (FDs):

- Course → Instructor  (Each course has one instructor)
- Instructor → Course  Not always true
- Instructor → Room  (Each instructor has a fixed room)

Why It's in 3NF but Not in BCNF?

- All non-key attributes are dependent on keys, so it satisfies 3NF.
- But Instructor → Room violates BCNF because:
 - Instructor is not a candidate key, but it's a determinant.
- Hence, not in BCNF.

Fix: Decompose into BCNF

Split the table into two relations:

- ◆ Table 1: Instructor_Room

Instructor	Room
Alice	R1
Bob	R2

FD: Instructor → Room ✓ Instructor is the key here.

Fix: Decompose into BCNF

- ◆ **Table 2: Course_Instructor**

Course	Instructor
DBMS	Alice
OS	Bob

FD: $\text{Course} \rightarrow \text{Instructor}$ Course is the key here.

Higher Normal Forms (4NF, 5NF)

- Handle multi-valued dependencies and join dependencies.
- Rarely used in everyday business applications but important in theoretical or complex systems.

Example: Joining employee and department tables

dept_id	dept_name
1	HR
2	IT
3	Finance

Example: Joining employee and department tables

emp_id	emp_name	dept_id
101	Alice	2
102	Bob	1
103	Charlie	3
104	David	NULL

Create Tables with Constraints

-- 1. Create Department table

```
CREATE TABLE IF NOT EXISTS department (
    dept_id SERIAL PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL
);
```

-- 2. Create Employee table with Foreign Key referencing Department

```
CREATE TABLE IF NOT EXISTS employee (
    emp_id SERIAL PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    dept_id_fk INTEGER,
    CONSTRAINT fk_department
        FOREIGN KEY (dept_id_fk)
        REFERENCES department(dept_id)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

Create .sql file

- Sudo –i –u postgres
- Psql
- \i /home/ubuntu-user/Documents/Labs/Lab01/script.sql

Describe Table

- \d department

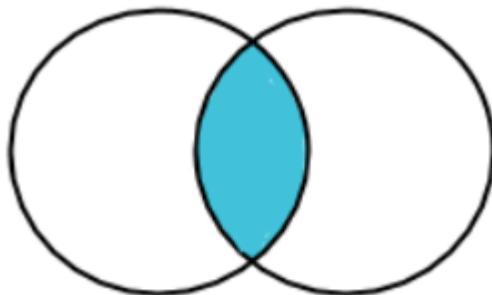
```
Table "public.department"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
dept_id | integer |          | not null | nextval('department_dept_id_seq'::regclas
s)
dept_name | text |          | not null |
Indexes:
"department_pkey" PRIMARY KEY, btree (dept_id)
```

Insert into table

- Insert into department(dept_name) values('HR');
- Insert into
employee(emp_name,dept_id_fk)values('param',1);

Joins

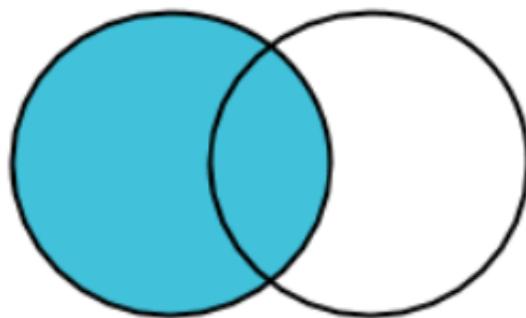
- **INNER JOIN:** Match rows in both tables



```
SELECT e.emp_id, e.emp_name, d.dept_name  
FROM employee e  
INNER JOIN department d ON e.dept_id_fk = d.dept_id;
```

Joins

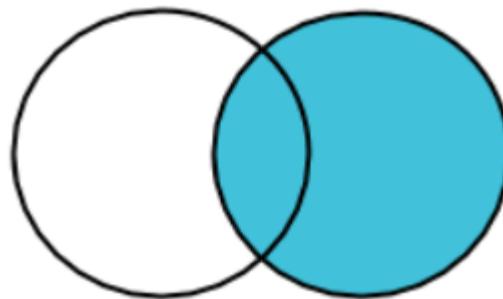
- **LEFT JOIN: All rows from left + matched rows from right**



```
SELECT e.emp_id, e.emp_name, d.dept_name  
FROM employee e  
LEFT JOIN department d ON e.dept_id = d.dept_id;
```

Joins

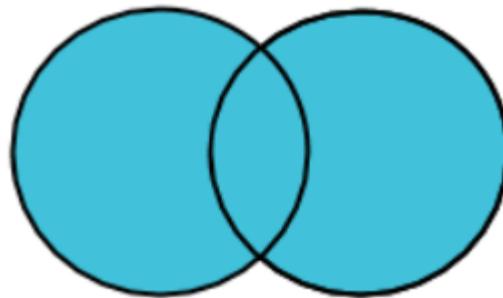
- **RIGHT JOIN: All rows from right + matched rows from left**



```
SELECT e.emp_id, e.emp_name, d.dept_name  
FROM employee e  
RIGHT JOIN department d ON e.dept_id = d.dept_id;
```

Joins

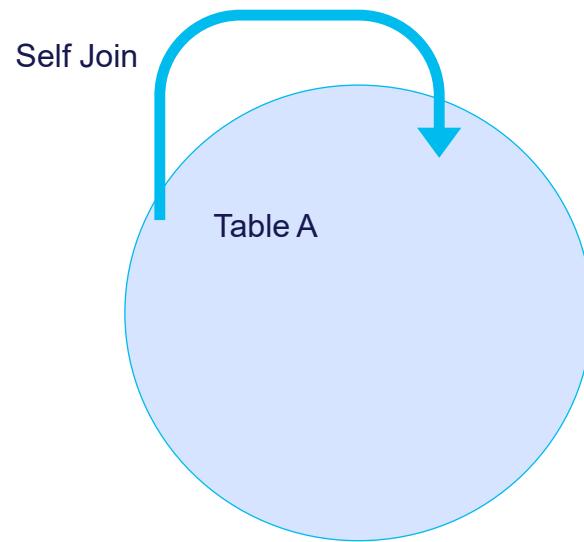
- **FULL OUTER JOIN: All rows from both tables, with NULLs where no match**



```
SELECT e.emp_id, e.emp_name, d.dept_name  
FROM employee e  
FULL OUTER JOIN department d ON e.dept_id = d.dept_id;
```

Joins

- **Self Join:** Join a table with itself (e.g., employee → manager)



Joins – Self Join

- **CREATE TABLE employee (**
- **emp_id SERIAL PRIMARY KEY,**
- **emp_name VARCHAR(100),**
- **manager_id INTEGER -- References another employee**
- **);**

Joins – Self Join

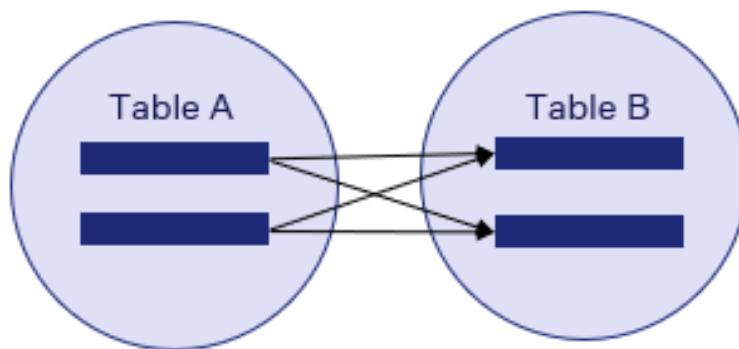
employee_id	employee_name	manager_name
1	Alice	NULL
2	Bob	Alice
3	Charlie	Alice
4	David	Bob

Joins – Self Join

- **SELECT**
- **e.emp_id AS employee_id,**
- **e.emp_name AS employee_name,**
- **m.emp_name AS manager_name**
- **FROM employee e**
- **LEFT JOIN employee m ON e.manager_id = m.emp_id;**

Joins

- **Cross Join: Cartesian product of two tables**



```
SELECT e.emp_name, d.dept_name  
FROM employee e  
CROSS JOIN department d;
```

Join with subquery

- **Get employees only from the IT department, using a subquery in the join:**
- SELECT e.emp_id, e.emp_name, d.dept_name
- FROM employee e
- JOIN (
 - SELECT dept_id, dept_name
 - FROM department
 - WHERE dept_name = 'IT'
 -) AS d ON e.dept_id = d.dept_id;

Joins Summary

Join Type	Matching Rows	Non-Matching Rows
INNER	<input checked="" type="checkbox"/>	<input type="checkbox"/>
LEFT	<input checked="" type="checkbox"/> (left)	NULLs from right
RIGHT	<input checked="" type="checkbox"/> (right)	NULLs from left
FULL	<input checked="" type="checkbox"/>	NULLs from both

Views

- A VIEW is a pseudo table in PostgreSQL; it is not a solid table but appears as an ordinary table to select.
- A view can also represent joined tables.
- It can contain all rows of a table or selected rows from one or more tables.
- A View simplifies users to perform the following aspects:
 - It structures data naturally and intuitively and makes it easy to find.
 - We can authorize permission to users over a view, which has a complete record that the users are authorized to see.
 - It restricts access to the data such that users can only see limited data instead of complete data.
 - A view provides a dependable layer, even the columns of necessary table modification.
 - It summarizes data from various tables to generate reports.
 - A view helps us to describe the difficulty of a statement as we can write a command of view based on a complex query with the help of a SELECT command.

Views

CREATE [OR REPLACE] VIEW view-name AS

SELECT column(s)

FROM table(s)

[WHERE condition(s)];

CREATE VIEW Price_View AS

SELECT id, price

FROM Price

WHERE price > 200;

Clear the screen

- `\! cls`

EXPLAIN – See the Query Plan

- Shows how PostgreSQL will execute a query, including:
 - Join types
 - Index usage
 - Cost estimates (not actual execution)

```
EXPLAIN SELECT * FROM employee WHERE dept_id = 2;
```

EXPLAIN ANALYZE – Run the Query and Show Actual Execution Details

- It **executes the query** and shows:
 - Actual time taken per step
 - Rows processed
 - Loops
 - Index usage
 - Total execution time

```
EXPLAIN ANALYZE SELECT * FROM employee WHERE dept_id = 2;
```

```
Seq Scan on employee (cost=0.00..25.00 rows=5 width=32) (actual  
time=0.021..0.023 rows=2 loops=1)
```

```
  Filter: (dept_id = 2)
```

```
  Rows Removed by Filter: 3
```

```
Planning Time: 0.100 ms
```

```
Execution Time: 0.050 ms
```

Key Terms Explained:

Term	Meaning
Seq Scan	Sequential scan of the table (may be slow on large tables)
Index Scan	Uses an index – usually faster
cost=X..Y	Estimated startup cost and total cost
rows=N	Estimated number of rows
actual time	Actual time to fetch rows
loops	How many times this step was executed
Rows Removed by Filter	Rows read but discarded by the WHERE clause

Best Practices

- Use EXPLAIN for safe preview.
- Use EXPLAIN ANALYZE to benchmark real execution.
- If the query is slow, look for:
 - Sequential scan instead of index scan
 - High number of loops
 - Large difference between estimated and actual rows

Explain and Analyze

```
public | profile      | table | enterprisedb
public | starrating   | table | admin
(5 rows)
```

```
accountdb=# explain analyze select * from producttransaction;
                                         QUERY PLAN
-----  
Seq Scan on producttransaction  (cost=0.00..1.02 rows=2 width=143) (actual time=10.385..10.388 rows=2 loops=1)
Planning Time: 7.873 ms
Execution Time: 29.357 ms
(3 rows)
```

```
accountdb=# explain analyze select * from customer;
                                         QUERY PLAN
-----  
Seq Scan on customer  (cost=0.00..1.01 rows=1 width=40) (actual time=0.022..0.024 rows=1 loops=1)
Planning Time: 0.103 ms
Execution Time: 0.044 ms
(3 rows)
```

```
accountdb=# explain analyze select * from globalcountryinfo;
                                         QUERY PLAN
-----  
Seq Scan on globalcountryinfo  (cost=0.00..1.01 rows=1 width=34) (actual time=27.068..27.072 rows=1 loops=1)
Planning Time: 372.275 ms
Execution Time: 27.098 ms
(3 rows)
```

```
accountdb=#
```

Materialized View

```
(3 rows)

accountdb=# explain analyze select * from customer;
                                QUERY PLAN
-----
Seq Scan on customer  (cost=0.00..1.01 rows=1 width=40) (actual time=0.022..0.024 rows=1 loops=1)
Planning Time: 0.103 ms
Execution Time: 0.044 ms
(3 rows)

accountdb=# explain analyze select * from globalcountryinfo;
                                QUERY PLAN
-----
Seq Scan on globalcountryinfo  (cost=0.00..1.01 rows=1 width=34) (actual time=27.068..27.072 rows=1 loops=1)
Planning Time: 372.275 ms
Execution Time: 27.098 ms
(3 rows)

accountdb=# create Materialized View mv_globalcountryinfo as select * from globalcountryinfo;
SELECT 1
accountdb=# explain analyze select * from mv_globalcountryinfo;
                                QUERY PLAN
-----
Seq Scan on mv_globalcountryinfo  (cost=0.00..23.10 rows=1310 width=34) (actual time=0.015..0.015 rows=1 loops=1)
Planning Time: 0.294 ms
Execution Time: 0.027 ms
(3 rows)

accountdb=#

```

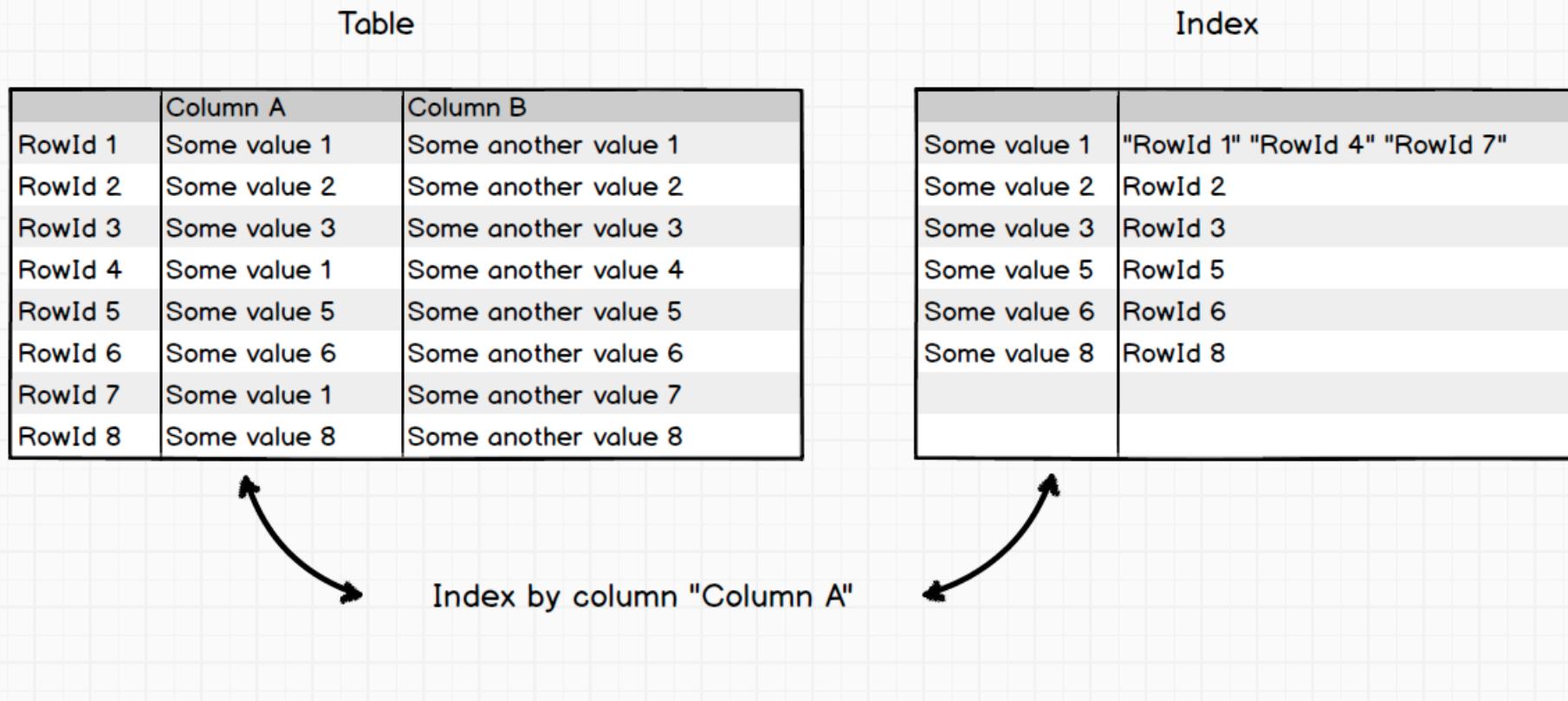
Refreshing a MATERIALIZED VIEW

```
EDB-PSQL - 13
accountdb=# refresh materialized view mv_globalcountryinfo;
REFRESH MATERIALIZED VIEW
accountdb=#
^
```

Postgres Index

- An Index is the structure or object by which we can retrieve specific rows or data faster.
- Indexes can be created using one or multiple columns or by using the partial data depending on your query requirement conditions.
- Index will create a pointer to the actual rows in the specified table.

Postgres Index



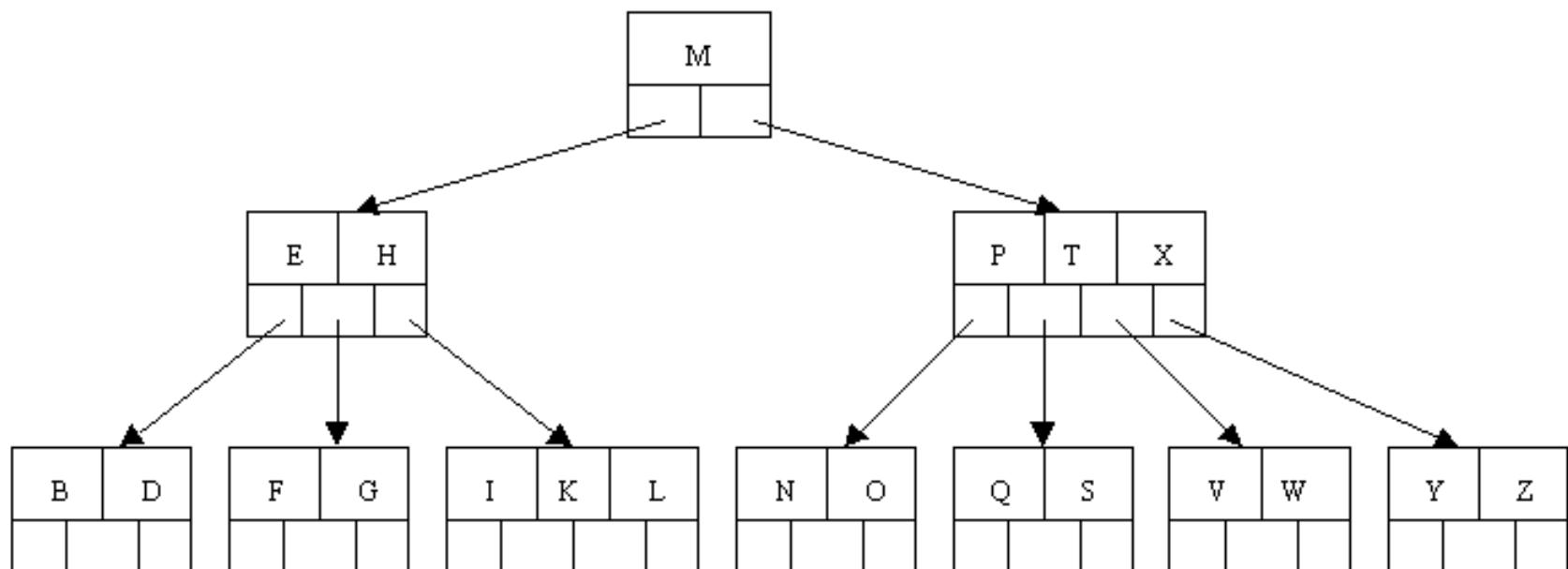
Postgres Index

- Types Of Indexes
 - PostgreSQL server provides following types of indexes, which each uses a different algorithm:
 - B-tree
 - Hash
 - GiST
 - SP-GiST
 - GIN
 - BRIN

B-tree index

- The most common and widely used index type is the B-tree index.
- This is the default index type for the CREATE INDEX command, unless you explicitly mention the type during index creation.
- If the indexed column is used to perform the comparison by using comparison operators such as <, <=, =, >=, and >, then the Postgres optimizer uses the index created by the B-tree option for the specified column.

B-tree index



B-tree index

- Advantages:
 - Retain sorting data
 - Support the search for the unary and binary predicates
 - Allow the entire sequence of data to estimate cardinality (number of entries) for the entire index (and therefore the table), range, and with arbitrary precision without scanning
- Disadvantages:
 - For their construction is require to perform a full sorting pairs (row, RowId) (slow operation)
 - Take up a lot of disk space. Index on unique “Integers” weights twice more as the column (because additionaly RowId need stored)
 - Recording unbalances tree constantly, and begins to store data sparsely, and the access time is increased by increasing the amount of disk information. What is why, B-Tree indexes require monitoring and periodic rebuilding

List indexes

- SELECT
- tablename,
- indexname,
- indexdef
- FROM
- pg_indexes
- WHERE
- schemaname = 'public'
- ORDER BY
- tablename,
- indexname;

List indexes

```
EJB-PSQL - 13
Output: transaction_id, transaction_date_time, amount, mode, status
Filter: (producttransaction.amount > 0)
Planning Time: 0.137 ms
Execution Time: 0.033 ms
(5 rows)

accountdb=# SELECT
accountdb-#   tablename,
accountdb-#   indexname,
accountdb-#   indexdef
accountdb-# FROM
accountdb-#   pg_indexes
accountdb-# WHERE
accountdb-#   schemaname = 'public'
accountdb-# ORDER BY
accountdb-#   tablename,
accountdb-#   indexname;
      tablename |      indexname |          indexdef
-----+-----+-----+
customer      | customer_pkey      | CREATE UNIQUE INDEX customer_pkey ON public.customer USING btree (customer_id)
globalcountryinfo | globalcountryinfo_pkey | CREATE UNIQUE INDEX globalcountryinfo_pkey ON public.globalcountryinfo USING btree (cou
ntrycode)
producttransaction | amount_index      | CREATE INDEX amount_index ON public.producttransaction USING btree (amount)
producttransaction | producttransaction_pkey | CREATE UNIQUE INDEX producttransaction_pkey ON public.producttransaction USING btree (t
ransaction_id)
profile         | profile_pkey       | CREATE UNIQUE INDEX profile_pkey ON public.profile USING btree (profile_id)
(5 rows)

accountdb=#

```

B-Tree Index

```
EDB-PSQL - 13
mode          | character varying(50)      |           |
status        | boolean                   |           |
Indexes:
"producttransaction_pkey" PRIMARY KEY, btree (transaction_id)

accountdb=# create index amount_index on producttransaction(amount);
CREATE INDEX
accountdb=# explain (analyze,verbose) select * from producttransaction;
               QUERY PLAN
-----
Seq Scan on public.producttransaction  (cost=0.00..1.02 rows=2 width=143) (actual time=0.016..0.017 rows=2 loops=1)
  Output: transaction_id, transaction_date_time, amount, mode, status
Planning Time: 0.626 ms
Execution Time: 0.032 ms
(4 rows)

accountdb=# \d producttransaction
              Table "public.producttransaction"
   Column    |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----+
transaction_id | bigint       |           | not null | nextval('producttransaction_transaction_id_seq'::regclass)
transaction_date_time | timestamp with time zone |           |           |
amount          | bigint       |           |           |
mode            | character varying(50) |           |           |
status          | boolean      |           |           |
Indexes:
"producttransaction_pkey" PRIMARY KEY, btree (transaction_id)
"amount_index" btree (amount)

accountdb=#
```

B-Tree Index

```
index_demo_=# explain (analyze,verbose) select * from public."Track" where "Name"='Princess of the Dawn'

                                         QUERY PLAN
-----
Bitmap Heap Scan on public."Track"  (cost=4.29..8.30 rows=1 width=71) (actual time=0.923..0.924 rows=1)

Output: "TrackId", "Name", "AlbumId", "Bytes", "UnitPrice"

Recheck Cond: (("Track"."Name")::text = 'Princess of the Dawn'::text)

Heap Blocks: exact=1

-> Bitmap Index Scan on "Track_Name_idx"  (cost=0.00..4.29 rows=1 width=0) (actual time=0.578..0.578 rows=1)

Index Cond: (("Track"."Name")::text = 'Princess of the Dawn'::text)

Planning time: 0.128 ms

Execution time: 0.953 ms

(8 rows)
```

B-Tree Index

```
EDB-PSQL - 13
accountdb=# refresh materialized view mv_globalcountryinfo;
REFRESH MATERIALIZED VIEW
accountdb=# explain (analyze,verbose) select * from globalcountryinfo;
                                         QUERY PLAN
-----
Seq Scan on public.globalcountryinfo  (cost=0.00..1.01 rows=1 width=34) (actual time=0.016..0.017 rows=1 loops=1)
  Output: countrycode, country_state
Planning Time: 0.074 ms
Execution Time: 0.037 ms
(4 rows)

accountdb=# explain (analyze,verbose) select * from mv_globalcountryinfo;
                                         QUERY PLAN
-----
Seq Scan on public.mv_globalcountryinfo  (cost=0.00..23.10 rows=1310 width=34) (actual time=0.017..0.017 rows=1 loops=1)
  Output: countrycode, country_state
Planning Time: 0.412 ms
Execution Time: 0.030 ms
(4 rows)

accountdb=# explain (analyze,verbose) select * from producttransaction;
                                         QUERY PLAN
-----
Seq Scan on public.producttransaction  (cost=0.00..1.02 rows=2 width=143) (actual time=0.019..0.021 rows=2 loops=1)
  Output: transaction_id, transaction_date_time, amount, mode, status
Planning Time: 0.079 ms
Execution Time: 0.033 ms
(4 rows)

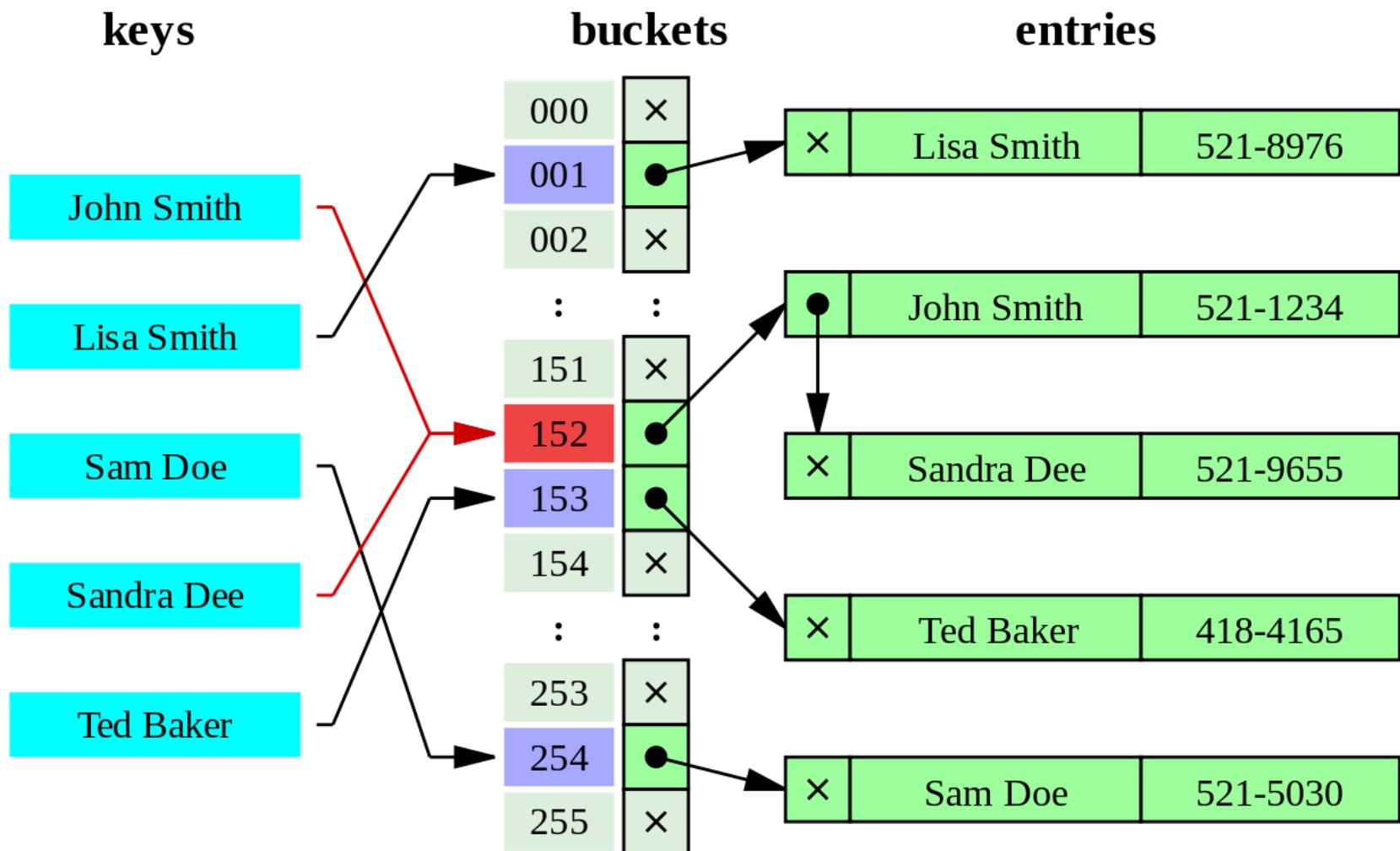
accountdb=#
```

B-Tree Index

Here is a query which can use the created B-tree index and then again try to select the data with the same **WHERE** clause:

```
index_demo_=# create index "Track_Name_idx" on public."Track" ("Name");  
CREATE INDEX  
index_demo_=#  
index_demo_=# explain (analyze,verbose) select * from public."Track" where "Name"='Princess of the Dawn'  
QUERY PLAN  
-----  
Index Scan using "Track_Name_idx" on public."Track" (cost=0.28..8.30 rows=1 width=71) (actual time=0.000..0.000 rows=1 loops=1)  
Output: "TrackId", "Name", "AlbumId", "MediaTypeId", "GenreId", "Composer", "Milliseconds", "Bytes", "UnitPrice"  
Index Cond: (("Track"."Name")::text = 'Princess of the Dawn'::text)  
Planning time: 0.151 ms  
Execution time: 0.571 ms  
(5 rows)
```

Hash index



Hash index

- Advantages:
 - Very fast search $O(1)$
 - Stability - the index does not need to be rebuild
- Disadvantages:
- Hash is very sensitive to collisions. In the case of “bad” data distribution, most of the entries will be concentrated in a few buckets, and in fact the search will occur through collision resolution.

Hash index

- The Hash index can be used only if the equality condition = is being used in the query.

```
index_demo_=# create index "Track_Name_idx" on public."Track" using HASH ("Name");
CREATE INDEX

index_demo_=# \d+ public."Track"

                                         Table "public.Track"

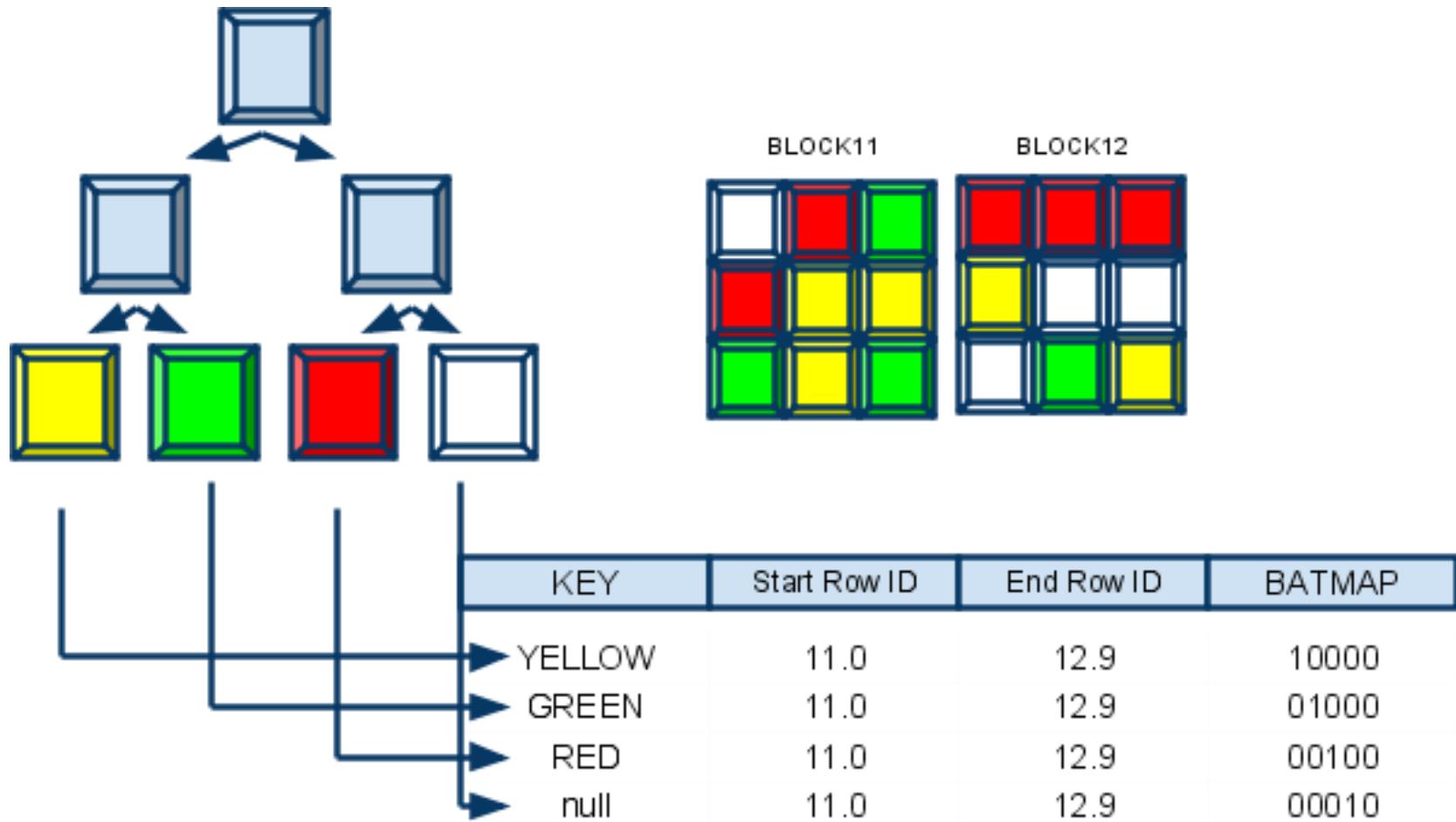
Column |      Type       | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+-----+
TrackId | integer      |           | not null |        | plain   |             |
Name    | character varying(200) |           | not null |        | extended |             |
AlbumId | integer      |           |           |        | plain   |             |
MediaTypeId | integer     |           |           | not null |        | plain   |             |
GenreId  | integer      |           |           |        | plain   |             |
Composer | character varying(220) |           |           |        | extended |             |
Milliseconds | integer    |           |           | not null |        | plain   |             |
Bytes    | integer      |           |           |        | plain   |             |
UnitPrice | numeric(10,2) |           |           | not null |        | main   |             |

Indexes:
"PK_Track" PRIMARY KEY, btree ("TrackId")
"Track_Name_idx" hash ("Name")
```

Bitmap index

- Bitmap index create a separate bitmap (a sequence of 0 and 1) for each possible value of the column, where each bit corresponds to a string with an indexed value. Bitmap indexes are optimal for data where bit unique values (example, gender field).

Bitmap index



Bitmap index

- Advantages:
 - Compact representation (small amount of disk space)
 - Fast reading and searching for the predicate “is”
 - Effective algorithms for packing masks (even more compact representation, than indexed data)
- Disadvantages:
 - You can not change the method of encoding values in the process of updating the data. From this it follows that if the distribution data has changed, it is required the index to be completely rebuild

GiST index (Generalized Search Tree index)

- GiST stands for Generalized Search Tree Index.
- The main point of GiST is to be able to index queries that simply are not indexable using B-tree.
- So GiST is useful when you have queries that are not btree-indexable.
- The column can be of tsvector or tsquery type.
- There is a well-known, and pretty fast, geographic extension to PostgreSQL — PostGIS, which uses GiST for its purposes. GiST indexes are what is used if you want to speed up full text searches.

GiST index (Generalized Search Tree index)

Create a table with some geographical data points such as circle or point and index it using GiST:

```
index_demo_=# create table geo_point(circle_details  circle);

CREATE TABLE

index_demo_=# create index ON geo_point using gist(circle_details);

CREATE INDEX

index_demo_=# \d+ geo_point

                                         Table "public.geo_point"

Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
circle_details | circle |        |        | plain |        |

Indexes:

"geo_point_circle_details_idx" gist (circle_details)

index_demo_=#
```

GiST index (Generalized Search Tree index)

Create a table with some geographical data points such as circle or point and index it using GiST:

```
index_demo_=# create table geo_point(circle_details  circle);

CREATE TABLE

index_demo_=# create index ON geo_point using gist(circle_details);

CREATE INDEX

index_demo_=# \d+ geo_point

                                         Table "public.geo_point"

Column |  Type   | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+
circle_details | circle |        |        | plain |        |

Indexes:

"geo_point_circle_details_idx" gist (circle_details)

index_demo_=#
```

SP-GiST (Space-Partitioned GiST)

- SP-GiST stands for “space-partitioned GiST.”
- This index type supports non-balanced data structures such as quadtrees, k-d trees, and radix trees.

GIN (Generalized Inverted Index)

- GIN stands for “Generalized Inverted Index.” This are the “Inverted Indexes” that get used for multiple component values such as arrays.
- The standard distribution of PostgreSQL includes a GIN operator class for arrays, which supports indexed queries using these operators: <@, @>, =, and &&.

BRIN (Block Range index)

- BRIN stands for “Block Range index.” This featured index type was introduced with Postgres v 9.5.
- “Because a BRIN index is very small, scanning the index adds little overhead compared to a sequential scan, but may avoid scanning large parts of the table that are known not to contain matching tuples.”
- “The size of the block range is determined at index creation time by the `pages_per_range` storage parameter.
- The number of index entries will be equal to the size of the relation in pages divided by the selected value for `pages_per_range`.
- Therefore, the smaller the number, the larger the index becomes (because of the need to store more index entries), but at the same time the summary data stored can be more precise and more data blocks can be skipped during an index scan.”

BRIN (Block Range index)

- In a BRIN index, PostgreSQL reads your selected column's maximum and minimum values for each 8k-size page of stored data, and then stores that information (page number and minimum and maximum values of column) into the BRIN index.

BRIN (Block Range index)

- B-tree indexes have entries for each row in your table, duplicating the data in the indexed columns.
- This permits super-fast index-only scans, but can be wasteful of disk space.
- A B-tree index also stores the data in sorted order, which permits very fast lookups of single rows.
- It also stores the full location of each row in the table, which can require a lot of work when running queries that return many rows.
- However, a BRIN index contains only the minimum and maximum values contained in a group of database pages.
- Therefore, the reason for using the BRIN index is that this is a very straightforward way to optimize for speed, and the BRIN index only takes up very little space compared to a B-tree index.
- BRIN is designed for handling very large tables in which certain columns have some natural correlation with their physical location within the table.

BRIN (Block Range index)

- A good example of where you could use a BRIN index is for data you would consider immutable.
- For the example that follows, the BRIN index is 72kb in size, while the B-tree index is 676mb in size.
- Yet the performance is pretty much identical for selects.
-

BRIN (Block Range index)

```
index_demo_=# create index "Track_TrackId_idx" on public."Track" using brin("TrackId");
CREATE INDEX

index_demo_=# \d+ public."Track"

                                         Table "public.Track"

Column |      Type       | Collation | Nullable | Default | Storage | Stats target | Description
-----+-----+-----+-----+-----+-----+-----+-----+
TrackId | integer      |           | not null |        | plain   |             |
Name    | character varying(200) |           | not null |        | extended |             |
AlbumId | integer      |           |           | plain   |             |
MediaTypeId | integer     |           | not null |        | plain   |             |
GenreId  | integer      |           |           | plain   |             |
Composer | character varying(220) |           |           | extended |             |
Milliseconds | integer    |           | not null |        | plain   |             |
Bytes    | integer      |           |           | plain   |             |
UnitPrice | numeric(10,2) |           | not null |        | main   |             |

Indexes:

"PK_Track" PRIMARY KEY, btree ("TrackId")

"Track_TrackId_idx" brin ("TrackId")

index_demo_=# \di+
```

Partial Index

- The partial index is useful in case you have commonly used WHERE conditions which use constant values as follows:

```
SELECT *  
FROM table_name  
WHERE column_name = constant_value;
```

Partial Index

- You can optimize this query by creating an index for the active column as follows:
- `CREATE INDEX idx_customer_active`
- `ON customer(active);`
- This index fulfills its purpose, however, it includes many rows that are never searched, namely all the active customers.
- To define an index that includes only inactive customers, you use the following statement:
- **`CREATE INDEX idx_customer_inactive`**
- **`ON customer(active)`**
- **`WHERE active = 0;`**

Built-In Functions- String

Function	Description	Example
LENGTH(string)	Returns the number of characters	LENGTH('hello') → 5
LOWER(string)	Converts to lowercase	LOWER('HELLO') → 'hello'
UPPER(string)	Converts to uppercase	UPPER('hello') → 'HELLO'
INITCAP(string)	Capitalizes first letter of each word	INITCAP('hello world') → 'Hello World'
`TRIM([LEADING BTRIM(str, chars)	TRAILING Trims specified characters from both ends	BOTH] 'x' FROM str)` BTRIM('xyHelloxy', 'xy') → 'Hello'
CONCAT(str1, str2, ...)	Concatenates strings	CONCAT('Hello', '', 'World') → 'Hello World'
'		'

Built-In Functions- String

Function	Description	Example
SUBSTRING(str FROM x FOR y)	Extracts substring	SUBSTRING('abcdef' FROM 2 FOR 3) → 'bcd'
LEFT(str, n)	Leftmost n characters	LEFT('abcdef', 3) → 'abc'
RIGHT(str, n)	Rightmost n characters	RIGHT('abcdef', 3) → 'def'
REPLACE(str, from, to)	Replace substring	REPLACE('hello', 'l', 'x') → 'hexxo'
POSITION(substr IN str)	Position of substring	POSITION('e' IN 'hello') → 2
STRPOS(str, substr)	Alias for POSITION	STRPOS('hello', 'e') → 2
OVERLAY(str PLACING str2 FROM pos FOR len)	Replaces part of string	OVERLAY('abcdef' PLACING 'xyz' FROM 2 FOR 3) → 'axyzef'

Built-In Functions- String

Function	Description	Example
REVERSE(str)	Reverses a string	REVERSE('abc') → 'cba'
LPAD(str, len, padstr)	Pads left side	LPAD('123', 5, '0') → '00123'
RPAD(str, len, padstr)	Pads right side	RPAD('123', 5, '0') → '12300'
REGEXP_REPLACE(str, pattern, replacement)	Regex-based replacement	REGEXP_REPLACE('abc123', '[0-9]', '', 'g') → 'abc'
REGEXP_MATCHES(str, pattern)	Returns regex match array	REGEXP_MATCHES('abc123', '[a-z]+') → {abc}

```
SELECT LENGTH('PostgreSQL'), LOWER('HELLO'), SUBSTRING('abcdef'  
FROM 2 FOR 3), REPLACE('banana', 'a', 'x'), LPAD('42', 5, '0');
```

Built in Date Functions

Function	Description	Example
CURRENT_DATE	Returns today's date	SELECT CURRENT_DATE;
CURRENT_TIME	Returns current time	SELECT CURRENT_TIME;
CURRENT_TIMESTAMP or NOW()	Returns current date and time	SELECT NOW();
AGE(timestamp)	Interval between now and given timestamp	AGE('2000-01-01')
AGE(timestamp1, timestamp2)	Interval between two timestamps	AGE('2025-01-01', '2000-01-01')
DATE_PART(field, source)	Extract part of a date (year, month, etc.)	DATE_PART('year', NOW())
EXTRACT(field FROM source)	Same as DATE_PART	EXTRACT(DAY FROM NOW())
DATE_TRUNC(field, source)	Truncates date/time to a unit	DATE_TRUNC('month', NOW()) → 2025-07-01 00:00:00
TO_CHAR(timestamp, format)	Format a timestamp	TO_CHAR(NOW(), 'YYYY-MM-DD')
TO_DATE(text, format)	Convert string to date	TO_DATE('2025-07-20', 'YYYY-MM-DD')

Built in Date Functions

Function	Description	Example
TO_TIMESTAMP(text, format)	Convert string to timestamp	TO_TIMESTAMP('2025-07-20 15:00', 'YYYY-MM-DD HH24:MI')
CLOCK_TIMESTAMP()	Returns actual current time (not cached)	SELECT CLOCK_TIMESTAMP();
STATEMENT_TIMESTAMP()	Timestamp when current statement started	SELECT STATEMENT_TIMESTAMP();
TRANSACTION_TIMESTAMP()	Timestamp when transaction started	SELECT TRANSACTION_TIMESTAMP();
JUSTIFY_DAYS(interval)	Converts excess days into months	JUSTIFY_DAYS(INTERVAL '90 days')
JUSTIFY_INTERVAL(interval)	Normalizes months and days	JUSTIFY_INTERVAL(INTERVAL '1 year 25 months')

```
-- Add 7 days to today
```

```
SELECT CURRENT_DATE + INTERVAL '7 days';
```

```
-- Subtract 1 month
```

```
SELECT CURRENT_DATE - INTERVAL '1 month';
```

Date Functions

- SELECT
- EXTRACT(YEAR FROM CURRENT_DATE) AS year,
- EXTRACT(MONTH FROM CURRENT_DATE) AS month,
- EXTRACT(DAY FROM CURRENT_DATE) AS day;

Date Functions

- `SELECT '2025-01-01'::date < '2025-12-31'::date;`
- -- returns true
- `SELECT TO_CHAR(NOW(), 'YYYY-MM-DD HH24:MI:SS');`
- -- Output: '2025-07-20 18:34:12'

Built-in Functions - JSON/JSONB Functions in PostgreSQL

Feature	JSON	JSONB
Storage	As text	Binary format
Speed	Slower to query	Faster and indexed
Ordering	Preserved	Not preserved
Duplicate Keys	Allowed	Deduplicated (last wins)

Built-in Functions - JSON/JSONB Functions in PostgreSQL

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name TEXT,
    data JSONB -- store nested info
);
```

```
INSERT INTO employees (name, data) VALUES
('Alice', '{"age": 30, "skills": ["SQL", "Python"]}'),
('Bob', '{"age": 25, "skills": ["Java", "React"]}');
```

Built-in Functions - JSON/JSONB Functions in PostgreSQL

Operator / Function	Description	Example
->	Get JSON object field by key (returns JSON)	data -> 'age'
->>	Get JSON object field as text	data ->> 'age'
#>	Get nested JSON object	data #> '{skills,0}'
#>>	Get nested JSON as text	data #>> '{skills,0}'
@>	Does JSONB contain the specified key/value?	data @> '{"age": 30}'
<@	Is JSONB contained in another?	'{ "age": 30 }' <@ data
?	Does key exist?	data ? 'age'
`?	`	Any of the keys exist?
?&	All keys exist?	data ?& array['age', 'skills']
jsonb_array_elements()	Expands JSON array to rows	see below
jsonb_each_text()	Extracts key-value pairs as text	see below

Built-in Functions - JSON/JSONB Functions in PostgreSQL

```
SELECT name, (data ->> 'age')::int AS age FROM employees;
```

```
SELECT * FROM employees WHERE data @> '{"age": 30}';
```

```
SELECT name, jsonb_array_elements_text(data->'skills') AS skill  
FROM employees;
```

```
SELECT name, key, value  
FROM employees, jsonb_each_text(data);
```

```
CREATE INDEX idx_employee_data ON employees USING GIN  
(data);
```

Moving Data the Copy command

- import and export data using CSV files in PostgreSQL
- 1. Creating a .csv file
- 2. Creating the table structure
- 3. Importing from a psql prompt
- 4. Importing from a shell prompt
- 5. Exporting data to a .csv file

Moving Data the Copy command

- Creating a .csv file
- The first step is to create a .csv file. You can use any existing file, or you can use the data below that comprises a basic .csv file:
- Capital_City,State,Abbreviation,Zip_Code
- Montgomery,Alabama,AL,36104
- Juneau,Alaska,AK,99801
- Phoenix,Arizona,AZ,85001
- Little Rock,Arkansas,AR,72201

Moving Data the Copy command

- CREATE TABLE usa
 - (Capital_city varchar,
 - State varchar,
 - Abbreviation varchar(2),
 - zip_code numeric(5)
-);

Importing from a psql prompt

- \COPY <table name> FROM 'location + file_name'
DELIMITER ',' CSV HEADER;
- To understand the execution steps, let's look at each item in this command:
- \COPY: This is the command to copy the record to / from the .csv file.
- <table name>: Provides the table name where you want to import the data.
- FROM: Specifies that we are going to import from a file (we will also be using TO in order to export it to a file at a later stage).

Importing from a psql prompt

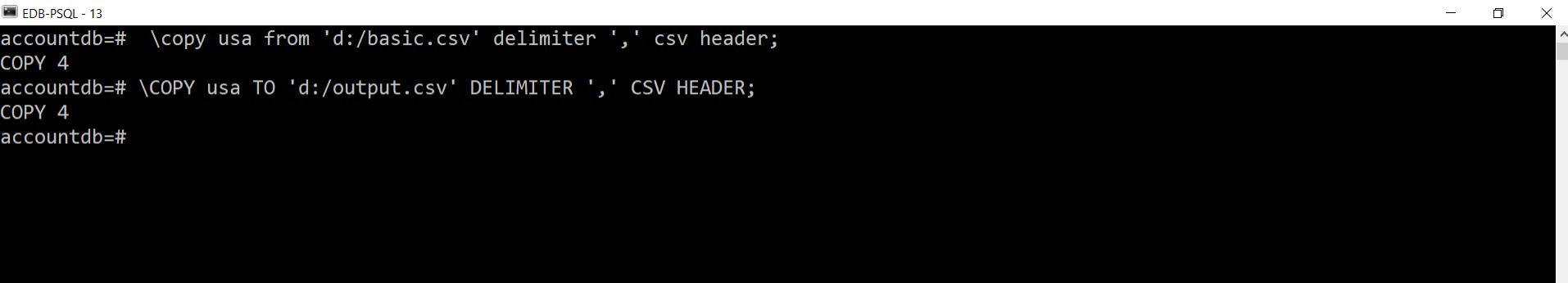
- 'location + file_name': An absolute path to the file (make sure you have read access to the file).
- DELIMITER ',': Specifies the delimiter, which in our case is a comma: ','.
- CSV Specifies the file type from which we are going to import.
- HEADER Signifies that we have a header row in our .csv file and while importing we should ignore the first row (similarly, while exporting we can use this to specify whether we want to include or exclude the header file).

Importing from a psql prompt

```
EDB-PSQL - 13
accountdb=# \copy usa from 'd:/basic.csv' delimiter ',' csv header;
COPY 4
accountdb=#

```

Exporting data to a .csv file



```
EDB-PSQL - 13
accountdb=# \copy usa from 'd:/basic.csv' delimiter ',' csv header;
COPY 4
accountdb=# \COPY usa TO 'd:/output.csv' DELIMITER ',' CSV HEADER;
COPY 4
accountdb=#

```

Understanding And Reading the PostgreSQL System Catalog (Meta Data)

- The PostgreSQL System Catalog is a schema with tables and views that contain metadata about all the other objects inside the database and more.
- With it, we can discover when various operations happen, how tables or indexes are accessed, and even whether or not the database system is reading information from memory or needing to fetch data from disk.

Understanding And Reading the PostgreSQL System Catalog (Meta Data)

- PostgreSQL stores the metadata information about the database and cluster in the schema ‘`pg_catalog`’.
- This information is partially used by PostgreSQL itself to keep track of things itself, but it also is presented so external people / processes can understand the inside of the databases too.

Understanding And Reading the PostgreSQL System Catalog (Meta Data)

- Database Information
- General database info is stored in pg_database and statistics are stored in pg_stat_database.
- `SELECT oid, * FROM pg_database WHERE datname = 'accountdb';`
- `SELECT * FROM pg_stat_database WHERE datname = 'accountdb';`

Understanding And Reading the PostgreSQL System Catalog (Meta Data)

- Database Activity and Locks
- There are two views that show current user activity, pg_stat_activity and pg_locks.
- When queried, these show information about current connections to the databases, and what kind of locks they have on what relations.
- `SELECT * FROM pg_stat_activity;`
- `SELECT * FROM pg_locks;`

Understanding And Reading the PostgreSQL System Catalog (Meta Data)

- `SELECT * FROM pg_stat_user_tables WHERE schemaname = 'public' AND relname = 'history';`

Understanding And Reading the PostgreSQL System Catalog (Meta Data)

- Table Vacuum Activity
- Table maintenance is done through either VACUUM or AUTOVACUUM, and statistics are gathered through ANALYZE or AUTOANALYZE.
- The next four columns contain the dates for when each of these operations were last run: ‘last_vacuum’, ‘last_autovacuum’, ‘last_analyze’, ‘last_autoanalyze’.
- We also have four more convenient columns that simply count how many times the previous actions occur.
- Using these, we can see which tables get the most activity: ‘vacuum_count’, ‘autovacuum_count’, ‘analyze_count’, and ‘autoanalyze_count’.

Understanding And Reading the PostgreSQL System Catalog (Meta Data)

- `SELECT * FROM pg_statio_user_tables WHERE schemaname = 'public' AND relname = history;`
- Index Meta data
- `SELECT * FROM pg_stat_user_indexes WHERE indexrelname = 'history_pkey';`

How to use table partitioning to scale PostgreSQL

- 1. Benefits of partitioning
- 2. When to use partitioning
- 3. How to use partitioning
 - a. List partition
 - b. Range partition
 - c. Hash partition
 - d. Multilevel partition
- 4. Limitations

Partitioning

- With huge data being stored in databases, performance and scaling are two main factors that are affected.
- As table size increases with data load, more data scanning, swapping pages to memory, and other table operation costs also increase.
- Partitioning may be a good solution, as It can help divide a large table into smaller tables and thus reduce table scans and memory swap problems, which ultimately increases performance.
- Partitioning helps to scale PostgreSQL by splitting large logical tables into smaller physical tables that can be stored on different storage media based on uses.
- Users can take better advantage of scaling by using declarative partitioning along with foreign tables using `postgres_fdw`.

Benefits of Partitioning

- PostgreSQL declarative partitioning is highly flexible and provides good control to users.
- Users can create any level of partitioning based on need and can modify, use constraints, triggers, and indexes on each partition separately as well as on all partitions together.
- Query performance can be increased significantly compared to selecting from a single large table.
- Partition-wise-join and partition-wise-aggregate features increase complex query computation performance as well.
- Bulk loads and data deletion can be much faster, as based on user requirements these operations can be performed on individual partitions.
- Each partition can contain data based on its frequency of use and so can be stored on media that may be cheaper or slower for low-use data.

When to use partitioning

- Most benefits of partitioning can be enjoyed when a single table is not able to provide them.
- So we can say that if a lot of data is going to be written on a single table at some point, users need partitioning.
- Apart from data, there may be other factors users should consider, like update frequency of the data, use of data over a time period, how small a range data can be divided, etc.
- With good planning and taking all factors into consideration, table partitioning can give a great performance boost and scale your PostgreSQL to larger datasets.

How to use partitioning

- As of PostgreSQL 12 release List, Range, Hash and combinations of these partition methods at different levels are supported.

LIST PARTITION

- A list partition is created with predefined values to hold in a partitioned table.
- A default partition (optional) holds all those values that are not part of any specified partition.
- `CREATE TABLE customers (id INTEGER, status TEXT, arr NUMERIC) PARTITION BY LIST(status);`
- `CREATE TABLE cust_active PARTITION OF customers FOR VALUES IN ('ACTIVE');`
- `CREATE TABLE cust_archived PARTITION OF customers FOR VALUES IN ('EXPIRED');`
- `CREATE TABLE cust_others PARTITION OF customers DEFAULT;`

RANGE PARTITION

- A range partition is created to hold values within a range provided on the partition key.
- Both minimum and maximum values of the range need to be specified, where minimum value is inclusive and maximum value is exclusive.
- `CREATE TABLE customers (id INTEGER, status TEXT, arr NUMERIC) PARTITION BY RANGE(arr);`
- `CREATE TABLE cust_arr_small PARTITION OF customers FOR VALUES FROM (MINVALUE) TO (25);`

RANGE PARTITION

- CREATE TABLE cust_arr_medium PARTITION OF customers FOR VALUES FROM (25) TO (75);
- CREATE TABLE cust_arr_large PARTITION OF customers FOR VALUES FROM (75) TO (MAXVALUE);

HASH PARTITION

- A hash partition is created by using modulus and remainder for each partition, where rows are inserted by generating a hash value using these modulus and remainders.
- `CREATE TABLE customers (id INTEGER, status TEXT, arr NUMERIC) PARTITION BY HASH(id);`
- `CREATE TABLE cust_part1 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder 0);`

HASH PARTITION

- CREATE TABLE cust_part2 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder 1);
- CREATE TABLE cust_part3 PARTITION OF customers FOR VALUES WITH (modulus 3, remainder 2);

MULTILEVEL PARTITION

- PostgreSQL multilevel partitions can be created up to N levels.
- Partition methods LIST-LIST, LIST-RANGE, LIST-HASH, RANGE-RANGE, RANGE-LIST, RANGE-HASH, HASH-HASH, HASH-LIST, and HASH-RANGE can be created in PostgreSQL declarative partitioning.

MULTILEVEL PARTITION

- CREATE TABLE customers (id INTEGER, status TEXT, arr NUMERIC) PARTITION BY LIST(status);
- CREATE TABLE cust_active PARTITION OF customers FOR VALUES IN ('ACTIVE','RECURRING','REACTIVATED') PARTITION BY RANGE(arr);
- CREATE TABLE cust_arr_small PARTITION OF cust_active FOR VALUES FROM (MINVALUE) TO (101) PARTITION BY HASH(id);

MULTILEVEL PARTITION

- CREATE TABLE cust_arr_small PARTITION OF cust_active FOR VALUES FROM (MINVALUE) TO (101) PARTITION BY HASH(id);
- CREATE TABLE cust_part11 PARTITION OF cust_arr_small FOR VALUES WITH (modulus 2, remainder 0);
- CREATE TABLE cust_part12 PARTITION OF cust_arr_small FOR VALUES WITH (modulus 2, remainder 1);
- CREATE TABLE cust_other PARTITION OF customers DEFAULT PARTITION BY RANGE(arr);

MULTILEVEL PARTITION

- CREATE TABLE cust_arr_large PARTITION OF cust_other FOR VALUES FROM (101) TO (MAXVALUE) PARTITION BY HASH(id);
- CREATE TABLE cust_part21 PARTITION OF cust_arr_large FOR VALUES WITH (modulus 2, remainder 0);
- postgres=# CREATE TABLE cust_part22 PARTITION OF cust_arr_large FOR VALUES WITH (modulus 2, remainder 1);

Limitations

- Partitioning was introduced in PostgreSQL 10 and continues to be improved and made more stable. Still, there are certain limitations that users may need to consider:
 - 1. Unique constraints on partitioned tables must include all the partition key columns. One work-around is to create unique constraints on each partition instead of a partitioned table.
 - 2. Partition does not support BEFORE ROW triggers on partitioned tables. If necessary, they must be defined on individual partitions, not the partitioned table.
 - 3. Range partition does not allow NULL values.
 - 4. PostgreSQL does not create a system-defined subpartition when not given it explicitly, so if a subpartition is present at least one partition should be present to hold values.
 - 5. In the case of HASH-LIST, HASH-RANGE, and HASH-HASH composite partitions, users need to make sure all partitions are present at the subpartition level as HASH can direct values at any partition based on hash value.

Postgres vs Oracle

	PostgreSQL Community	Oracle Standard Edition	Oracle Enterprise Edition
Single Database Edition Costs	\$0	\$17,500	\$47,500
Clustering / Replication	\$0	Not available	\$23,000
Advanced Security	\$0	Not available	\$15,000
Software Update License & Support (annual)	\$0	Not available	\$18,810
Total Cost	\$0	\$17,500	\$104,310

Postgres vs Oracle

	PostgreSQL	Oracle
Scalability	<p>PostgreSQL offers free scalability, and can scale up to millions of transactions per seconds.</p>	<p>Oracle Enterprise is recommended for high workloads which are highly scalable, but costly.</p>
Updates	<p>Since the last few years, new major PostgreSQL versions are released every year and minor versions with bug fixes are released every 3 months. One of the best things about Postgres is that the PostgreSQL Global Development Group announces the major and minor release dates in advance for the convenience of users and prospects.</p>	<p>New Oracle versions are generally available every 2-4 years.</p>

Postgres vs Oracle

Security	<p>PostgreSQL offers strong security capabilities through different authentication options (Host, LDAP, PAM and certificates authentication) and role-based access control (user-level, table-level and row-level). Data encryption can be achieved with advanced security plugins like pgcrypto which are available for free.</p>	<p>Oracle offers advanced security packages, but as a commercial database, they are available as an expensive add-on.</p>
Replication	<p>PostgreSQL supports native streaming replication and logical replication.</p>	<p>Oracle supported master-slave and master-master replication via Oracle Streams and Oracle multi-master in the older versions of Oracle Enterprise Edition, which has now been replaced by Oracle GoldenGate (a separately licensed application) for all types of data replications.</p>

Postgres vs Oracle

Partitioning	PostgreSQL supports declarative partitioning .	Oracle supports general horizontal partitioning supported by all RDBMS.
Cloud Deployments	Can be deployed on any cloud provider, with a variety of PostgreSQL hosting solutions available.	Can only be deployed on Oracle Cloud and other popular cloud providers, but users must Bring Your Own License (BYOL) or use on-demand licensing .

Postgres vs Oracle

	PostgreSQL	Oracle
Compatibility	PostgreSQL PL/pgSQL is compatible with other relational databases like Oracle which makes it relatively easy to move to PostgreSQL.	Oracle infrastructure does not offer strong compatibility with open source RDBMS.
Extensions	PostgreSQL offers many free open-source extensions, including: PostGIS - support for geospatial objects store and query. CitusDB - distributes data and queries horizontally across nodes. pg_repack - reorganizes tables online to reclaim storage.	Oracle offers commercial add-ons that are available for an additional license fee .

Postgres vs Oracle

Extensions	<p>PostgreSQL offers many free open-source extensions, including:PostGIS – support for geospatial objects store and query.CitusDB – distributes data and queries horizontally across nodes.pg_repack – reorganizes tables online to reclaim storage.</p>	<p>Oracle offers commercial add-ons that are available for an additional license fee.</p>
Tuning	<p>PostgreSQL offers more light-weight tuning capabilities, like their Query Optimizer, and DBaaS platforms like ScaleGrid offer advanced slow query analysis.</p>	<p>Oracle requires significantly more effort to install and configure due to the hundreds of tuning variables and complex system requirements. Most of the tuning capabilities provided from Automatic Workload Repository (AWR) and database advisers comes bundled with Oracle Enterprise Manager Database/Grid control that requires the Enterprise Edition.</p>

Postgres vs Oracle

Supported Operating Systems	<ul style="list-style-type: none">• AIX• HP-UX• Linux• OS X• Solaris• Windows• z/OS	<ul style="list-style-type: none">• FreeBSD• HP-UX• Linux• NetBSD• OpenBSD• OS X• Solaris• Unix• Windows
Supported Languages	<ul style="list-style-type: none">• C• C#• C++• Clojure• Cobol• Delphi• Eiffel• Erlang• Fortran• Groovy• Haskell• Java• JavaScript	<ul style="list-style-type: none">• .Net• C• C++• Delphi• Java• JavaScript (Node.js)

Backup and Restore

- PostgreSQL provides the pg_dump utility to help you back up databases.
- It generates a database file with SQL commands in a format that can be easily restored in the future.
- To back up a PostgreSQL database, start by logging into your database server, then switch to the Postgres user account, and run pg_dump as follows (replace tecmintdb with the name of the database you want to backup).
- By default, the output format is a plain-text SQL script file.

Backup and Restore

- Understand the Basic Syntax of pg_dump
- PostgreSQL comes with built-in utilities called pg_dump for easily creating and restoring backups.
- The basic syntax of pg_dump command is shown below:
- pg_dump [OPTION]... [DBNAME]
- A brief explanation of each option is shown below:
 - -d, –dbname=DATABASENAME : Used to specify the database that you want to back up.
 - -h, –host=HOSTNAME : Used to specify the hostname of your database server.
 - -U, –username=USERNAME : Used to specify the PostgreSQL username.
 - -w, –no-password : Used to ignore the password prompt.
 - -p, –port=PORT : Used to specify the PostgreSQL server port number.
 - -W, –password : Used to force password prompt.
 - –role=ROLENAMESPACE : SET ROLE before the dump.

Backup and Restore

- Backup
 - `pg_dump -d testdb -f testdb_backup.sql`
- Restore
 - `psql -d testdb -f testdb_backup.sql`
- If you want to stop the database restore process in case an error occurs, run the following command:
 - `psql -d testdb --set ON_ERROR_STOP=on -f testdb_backup.sql`

Backup and Restore

```
Administrator: Command Prompt
D:\PostgreSQL\13\bin>pg_dump.exe -U admin -d accountdb -f D:\account_dump.sql
Password:
D:\PostgreSQL\13\bin>
```

```
D:\Program Files\edb\as13\bin>psql.exe -U admin -d accountdb -f D:\account_dump.sql
Password for user admin:
SET
SET
SET
SET
SET
SET
set_config
-----
(1 row)

SET
SET
SET
SET
SET
SET
SET
CREATE TABLE
```

Admin Queries

```
SELECT pg_size.pretty(pg_total_relation_size('table_name')) AS total_size;
```

The screenshot shows a terminal window with two panes. The left pane, titled 'script.sql', contains the SQL query:

```
select pg_size.pretty(pg_total_relation_size
('sensors.observations')) as total_size;
```

The right pane, titled 'Query Results', displays the output of the query:

total_size
2352 kB

Below the results, there is a section titled 'Database Schema'.

Admin Queries

```
SELECT indexname, indexdef  
FROM pg_indexes  
WHERE tablename = 'sensors.observations';
```

Admin Queries - Size of Each Index on a Table

```
SELECT
    i.relname AS index_name,
    pg_size.pretty(pg_relation_size(i.oid)) AS index_size,
    pg_relation_size(i.oid) AS size_bytes
FROM
    pg_class t
JOIN
    pg_index ix ON t.oid = ix.indrelid
JOIN
    pg_class i ON i.oid = ix.indexrelid
WHERE
    t.relname = 'employee'
ORDER BY
    size_bytes DESC;
```

Admin Queries - Query: Table Data, Index, and Total Size

```
SELECT
    relname AS table_name,
    pg_size.pretty(pg_relation_size(relid)) AS data_size,
    pg_size.pretty(pg_indexes_size(relid)) AS index_size,
    pg_size.pretty(pg_total_relation_size(relid)) AS total_size
FROM
    pg_catalog.pg_statio_user_tables
WHERE
    relname = 'sensors.observations';
```

Backup and Restore All Databases

- You can back up all databases in PostgreSQL using pg_dumpall utility. The basic syntax to backup all databases as shown below:
 - pg_dumpall -f [alldatabase_backup.sql]
- For example, to back up all databases in PostgreSQL and generate a backup file named alldb_backup.sql, run the following command:
 - pg_dumpall -f alldb_backup.sql
- To restore all databases from a backup file named alldb_backup.sql, run the following command:
 - psql -f alldb_backup.sql

Backup and Restore Single Table

- PostgreSQL also allows you to back up a single table from the specific database. You can achieve this using the following syntax:
- `pg_dump -d [source-database] -t [table_name]-f [dbtable_backup.sql]`
- For example, to back up a table named mytab from the database named testdb and generate a backup file named `testdb_mytab.sql`, run the following command:
- `pg_dump -d testdb -t mytab -f testdb_mytab_backup.sql`
- If you want to restore this table from the backup file, run the following command:
- `psql -d testdb -f testdb_mytab_backup.sql`

Backup and Restore Compressed Data Store

- You can also back up the PostgreSQL database and compress it in .gz format to reduce the backup size.
- To take a back up of database named testdb and generate a compressed backup file named testdb_compressed.sql.gz, run the following command:

```
pg_dump -d testdb | gzip > testdb_compressed.sql.gz
```
- You can also restore the backup from the compressed file using the following command:

```
gunzip -c testdb_compressed.sql.gz | psql -d testdb
```

PL/PGSQL

- PL/pgSQL procedural language adds many procedural elements e.g., control structures, loops, and complex computations to extend standard SQL.
- It allows you to develop complex functions and stored procedures in PostgreSQL that may not be possible using plain SQL.
- Since PostgreSQL 9.0, PL/pgSQL is installed by default.

PL/PGSQL

- PL/pgSQL procedural language is similar to the Oracle PL/SQL. The following are reasons to learn PL/pgSQL:
 - PL/pgSQL is easy to learn and simple to use.
 - PL/pgSQL comes with PostgreSQL by default. The user-defined functions and stored procedures developed in PL/pgSQL can be used like any built-in functions and stored procedures.
 - PL/pgSQL inherits all user-defined types, functions, and operators.
 - PL/pgSQL has many features that allow you to develop complex functions and stored procedures.
 - PL/pgSQL can be defined to be trusted by the PostgreSQL database server.

PL/PGSQL

- PL/pgSQL is a procedural programming language for the PostgreSQL database system.
- PL/pgSQL allows you to extend the functionality of the PostgreSQL database server by creating server objects with complex logic.
- PL/pgSQL was designed to :
 - Create user-defined functions, stored procedures, and triggers.
 - Extend standard SQL by adding control structures such as if, case, and loop statements.
 - Inherit all user-defined functions, operators, and types.

Advantages of using PL/pgSQL

- SQL is a query language that allows you to query data from the database easily. However, PostgreSQL only can execute SQL statements individually.
- It means that you have multiple statements, you need to execute them one by one like this:
 - First, send a query to the PostgreSQL database server.
 - Next, wait for it to process.
 - Then, process the result set.
 - After that, do some calculations.
 - Finally, send another query to the PostgreSQL database server and repeat this process.

Advantages of using PL/pgSQL

- This process incurs the interprocess communication and network overheads.
- To resolve this issue, PostgreSQL uses PL/pgSQL.
 - PL/pgSQL wraps multiple statements in an object and store it on the PostgreSQL database server.
 - So instead of sending multiple statements to the server one by one, you can send one statement to execute the object stored in the server. This allows you to:
 - Reduce the number of round trips between the application and the PostgreSQL database server.
 - Avoid transferring the immediate results between the application and the server.

PostgreSQL PL/pgSQL disadvantages

- Besides the advantages of using PL/pgSQL, there are some caveats:
 - Slower in software development because PL/pgSQL requires specialized skills that many developers do not possess.
 - Difficult to manage versions and hard to debug.
 - May not be portable to other database management systems.

Dollar-Quoted String Constants

- In PostgreSQL, you use single quotes for a string constant like this:
 - select 'String constant';
- When a string constant contains a single quote ('), you need to escape it by doubling up the single quote. For example:
 - select 'I"m also a string constant';

Dollar-Quoted String Constants

- PostgreSQL version 8.0 introduced the dollar quoting feature to make string constants more readable.
- The following shows the syntax of the dollar-quoted string constants:
 - \$tag\$<string_constant>\$tag\$
- In this syntax, the tag is optional. It may contain zero or many characters.
 - Between the \$tag\$, you can place any string with single quotes ('') and backslashes (\).
 - For example:
 - select \$\$I'm a string constant that contains a backslash \\$\\$\n;

Dollar-Quoted String Constants

- In this example, we did not specify the tag between the two dollar signs(\$).
- The following example uses the dollar-quoted string constant syntax with a tag:
 - SELECT \$message\$I'm a string constant that contains a backslash \\$message\$;
 - In this example, we used the string message as a tag between the two dollar signs (\$)

Using dollar-quoted string constant in anonymous blocks

```
do
'declare
    film_count integer;
begin
    select count(*) into film_count
    from film;
    raise notice "The number of films: %", film_count;
end;'
```

Using dollar-quoted string constant in anonymous blocks

To avoid escaping every single quotes and backslashes, you can use the dollar-quoted string as follows

```
do
$$
declare
    film_count integer;
begin
    select count(*) into film_count
    from film;
    raise notice 'The number of films: %', film_count;
end;
$$
```

Using dollar-quoted string constant in anonymous blocks

The screenshot shows a PostgreSQL query editor interface. At the top, there's a navigation bar with links for Dashboard, Properties, SQL, Statistics, Dependencies, and Dependents. The connection information is displayed as accountdb/enterprisedb@EDB Postgres Advanced Server 13 *. Below the navigation bar is a toolbar with various icons for database management tasks. The main area is titled "Query Editor" and contains the following PostgreSQL code:

```
1 do
2 $$
3 declare
4     film_count integer;
5 begin
6     select count(*) into film_count
7     from films;
8     raise notice 'The number of films: %', film_count;
9 end;
10 $$
```

Below the code, there are tabs for Data Output, Explain, Messages, and Notifications. The Messages tab is currently selected, showing the output of the query:

NOTICE: The number of films: 3
DO

Query returned successfully in 1 secs 14 msec.

Using dollar-quoted string constants in functions

```
create function find_film_by_id(
```

```
    id int
```

```
) returns film
```

```
language sql
```

```
as
```

```
$$
```

```
    select * from film
```

```
        where film_id = id;
```

```
$$;
```

Using dollar-quoted string constants in functions

```
EDB-PSQL - 13
accountdb=# create function find_film_by_id(
accountdb(#   id int
accountdb(# ) returns films
accountdb# language sql
accountdb# as
accountdb# $$
accountdb$#   select * from films
accountdb$#   where film_id = id;
accountdb$# $$;
CREATE FUNCTION
```

The screenshot shows a PostgreSQL query editor interface. At the top, there's a toolbar with various icons for database management. Below the toolbar, the connection information is displayed: accountdb/enterprisedb@EDB Postgres Advanced Server 13. The main area is divided into two tabs: 'Query Editor' (which is currently selected) and 'Query History'. In the 'Query Editor' tab, the following SQL code is written:

```
1 select find_film_by_id(1) from dual;
```

At the bottom of the interface, there are several tabs: 'Data Output', 'Explain', 'Messages', and 'Notifications'. Under the 'Data Output' tab, the results of the query are shown in a table:

	find_film_by_id	films
1	(1,Joker)	

PL/pgSQL Block Structure

- PL/pgSQL is a block-structured language, therefore, a PL/pgSQL function or stored procedure is organized into blocks.

```
[ <<label>> ]
```

```
[ declare  
    declarations ]
```

```
begin  
    statements;
```

```
...
```

```
end [ label ];
```

PL/pgSQL Block Structure

- PL/pgSQL is a block-structured language, therefore, a PL/pgSQL function or stored procedure is organized into blocks.

```
[ <<label>> ]
```

```
[ declare  
    declarations ]
```

```
begin  
    statements;
```

```
...
```

```
end [ label ];
```

PL/pgSQL Block Structure

- Each block has two sections: declaration and body.
- The declaration section is optional while the body section is required.
- A block is ended with a semicolon (;) after the END keyword.
- A block may have an optional label located at the beginning and at the end.
- We use the block label when you want to specify it in the EXIT statement of the block body or when you want to qualify the names of variables declared in the block.
- The declaration section is where you declare all variables used within the body section.
- Each statement in the declaration section is terminated with a semicolon (;).
- The body section is where you place the code. Each statement in the body section is also terminated with a semicolon (;).

PL/pgSQL Block Structure

```
do $$  
  <<first_block>>  
declare  
    film_count integer := 0;  
begin  
    -- get the number of films  
    select count(*)  
    into film_count  
    from films;  
    -- display a message  
    raise notice 'The number of films is %', film_count;  
end first_block $$;
```

PL/pgSQL Block Structure

The screenshot shows a PostgreSQL query editor interface. At the top, there's a navigation bar with tabs: Dashboard, Properties, SQL, Statistics, Dependencies, Dependents, and a connection dropdown set to accountdb/enterprisedb@EDB Postgres Advanced Server 13 *. Below the navigation bar is a toolbar with various icons for file operations like Open, Save, Print, and a search/filter icon.

The main area is a Query Editor containing the following PL/pgSQL code:

```
1 do $$  
2 <<first_block>>  
3 declare  
4   film_count integer := 0;  
5 begin  
6   -- get the number of films  
7   select count(*)  
8   into film_count  
9   from films;  
10  -- display a message  
11  raise notice 'The number of films is %', film_count;  
12 end first_block $$;
```

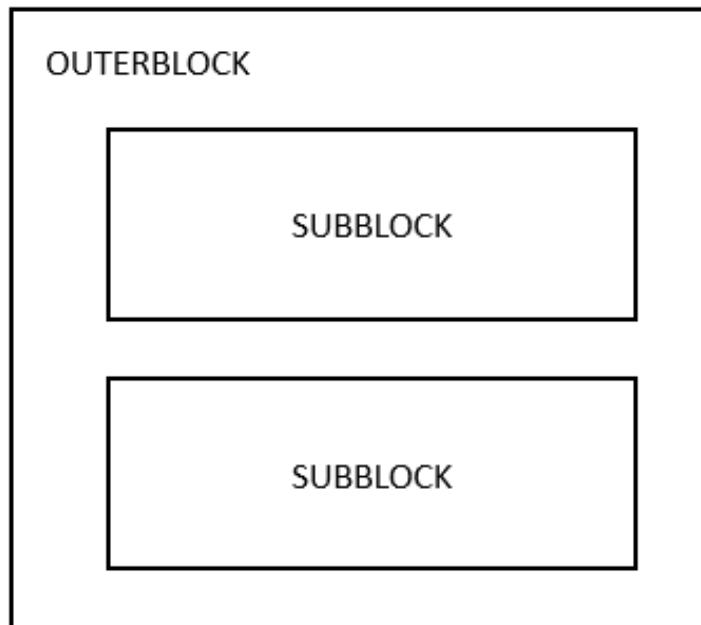
Below the code, there are tabs for Data Output, Explain, Messages, and Notifications. The Messages tab is currently selected, showing the output:

NOTICE: The number of films is 3
DO

Query returned successfully in 217 msec.

PL/pgSQL Subblocks

- PL/pgSQL allows you to place a block inside the body of another block.
- The block nested inside another block is called a subblock. The block that contains the subblock is referred to as an outer block.



Introduction to variables in PL/pgSQL

- A variable is a meaningful name of a memory location.
- A variable holds a value that can be changed through the block.
- A variable is always associated with a particular data type.
- `variable_name data_type [:= expression];`

Introduction to variables in PL/pgSQL

- First, specify the name of the variable.
- It is a good practice to assign a meaningful name to a variable.
- For example, instead of naming a variable `i` you should use `index` or `counter`.
- Second, associate a specific data type with the variable.
- The data type can be any valid data type such as `integer`, `numeric`, `varchar`, and `char`.
- Third, optionally assign a default value to a variable. If you don't do so, the initial value of the variable is `NULL`.

Introduction to variables in PL/pgSQL

- First, specify the name of the variable.
- It is a good practice to assign a meaningful name to a variable.
- For example, instead of naming a variable `i` you should use `index` or `counter`.
- Second, associate a specific data type with the variable.
- The data type can be any valid data type such as `integer`, `numeric`, `varchar`, and `char`.
- Third, optionally assign a default value to a variable. If you don't do so, the initial value of the variable is `NULL`.
- Note that you can use either `:=` or `=` assignment operator to initialize and assign a value to a variable.

Introduction to variables in PL/pgSQL

```
do $$  
declare  
    counter    integer := 1;  
    first_name varchar(50) := 'John';  
    last_name  varchar(50) := 'Doe';  
    payment    numeric(11,2) := 20.5;  
begin  
    raise notice '% % % has been paid % USD',  
        counter,  
        first_name,  
        last_name,  
        payment;  
end $$;
```

Variable initialization timing

```
do $$  
declare  
    created_at time := now();  
begin  
    raise notice '%', created_at;  
    perform pg_sleep(10);  
    raise notice '%', created_at;  
end $$;
```

Variable initialization timing

The screenshot shows a PostgreSQL query editor interface. At the top, it displays the connection details: accountdb/enterprisedb@EDB Postgres Advanced Server 13. Below the connection bar, there are two tabs: "Query Editor" (which is selected) and "Query History". The main area contains a code snippet:

```
1 do $$  
2 declare  
3     created_at time := now();  
4 begin  
5     raise notice '%', created_at;  
6     perform pg_sleep(10);  
7     raise notice '%', created_at;  
8 end $$;  
9
```

Below the code, there are four tabs: "Data Output", "Explain", "Messages" (which is underlined, indicating it is active), and "Notifications". Under the "Messages" tab, the following output is shown:

```
NOTICE:  20:46:10.686542  
NOTICE:  20:46:10.686542  
DO
```

At the bottom of the interface, a message states: "Query returned successfully in 10 secs 636 msec."

Copying data types

- The %type provides the data type of a table column or another variable.
- Typically, you use the %type to declare a variable that holds a value from the database or another variable.
 - `variable_name table_name.column_name%type;`
- And the following shows how to declare a variable with the data type of another variable:
 - `variable_name variable%type;`

Copying data types

```
do $$  
declare  
    film_title films.title%type;  
    featured_title film_title%type;  
begin  
    -- get title of the film id 100  
    select title  
    from films  
    into film_title  
    where film_id = 1;  
  
    -- show the film title  
    raise notice 'Film title id 100: %s', film_title;  
end; $$
```

Variables in block and subblock

- When you declare a variable in a subblock which has the same name as another variable in the outer block, the variable in the outer block is hidden in the subblock.

```
do $$  
  <<outer_block>>  
  declare  
    counter integer := 0;  
  begin  
    counter := counter + 1;  
    raise notice 'The current value of the  
    counter is %', counter;  
  
    declare  
      counter integer := 0;  
    begin  
      counter := counter + 10;  
      raise notice 'Counter in the subblock is  
      %', counter;  
      raise notice 'Counter in the outer block  
      is %', outer_block.counter;  
    end;  
  
    raise notice 'Counter in the outer block is  
    %', counter;  
  end outer_block $$;
```

PL/pgSQL row types example

```
do $$  
declare  
    selected_film films%rowtype;  
begin  
    -- select film with id 1  
    select *  
    from films  
    into selected_film  
    where film_id = 1;  
  
    -- show the number of actor  
    raise notice 'The film name is % %',  
        selected_film.film_id,  
        selected_film.title;  
end; $$
```

PL/pgSQL Record Types

```
do
$$
declare
    rec record;
begin
    -- select the film
    select film_id, title
        into rec
    from films
    where film_id = 1;

    raise notice '% %', rec.film_id, rec.title;

end;
$$
```

Using record variables in the for loop statement

```
do
$$
declare
    rec record;
begin
    for rec in select title
        from films
        where film_id > 0
        order by title
    loop
        raise notice '%', rec.title;
    end loop;
end;
$$
```

PL/pgSQL Constants

- do \$\$
- declare
 - vat constant numeric := 0.1;
 - net_price numeric := 20.5;
- begin
 - raise notice 'The selling price is %', net_price * (1 + vat);
- end \$\$;

PL/pgSQL Errors and Messages

- Reporting messages
- To raise a message, you use the `raise` statement as follows:
 - `raise level format;`
- Following the `raise` statement is the `level` option that specifies the error severity.
- PostgreSQL provides the following levels:
 - `debug`
 - `log`
 - `notice`
 - `info`
 - `warning`
 - `exception`

Reporting Messages

```
do $$  
begin  
    raise info 'information message %', now() ;  
    raise log 'log message %', now();  
    raise debug 'debug message %', now();  
    raise warning 'warning message %', now();  
    raise notice 'notice message %', now();  
end $$;
```

Raising errors

- To raise an error, you use the exception level after the raise statement.
- Note that raise statement uses the exception level by default.

Raising errors

- Besides raising an error, you can add more information by using the following additional clause:
- using option = expression
- The option can be:
 - message: set error message
 - hint: provide the hint message so that the root cause of the error is easier to be discovered.
 - detail: give detailed information about the error.
 - errcode: identify the error code, which can be either by condition name or directly five-character SQLSTATE code. Please refer to the table of error codes and condition names.

Raising Errors

```
do $$  
declare  
    email varchar(255) := 'info@postgresqltutorial.com';  
begin  
    -- check email for duplicate  
    -- ...  
    -- report duplicate email  
    raise exception 'duplicate email: %', email  
        using hint = 'check the email again';  
end $$;
```

Assert Statement

```
do $$  
declare  
    film_count integer;  
begin  
    select count(*)  
    into film_count  
    from films;  
  
    assert film_count > 1000, '1000 Film not found, check the  
film table';  
end$$;
```

PL/pgSQL IF Statement

- The if statement determines which statements to execute based on the result of a boolean expression.
- PL/pgSQL provides you with three forms of the if statements.
 - if then
 - if then else
 - if then elsif

PL/pgSQL IF Statement

- PL/pgSQL if-then statement
- The following illustrates the simplest form of the if statement:
 - if condition then
 - statements;
 - end if;

PL/pgSQL IF Statement

```
do $$  
declare  
    selected_film films%rowtype;  
    input_film_id films.film_id%type := 0;  
  
begin  
    select * from films  
    into selected_film  
    where film_id = input_film_id;  
    if not found then  
        raise notice'EThe film % could not be found',  
            input_film_id;  
    end if;  
end $$
```

PL/pgSQL if-then-else statement

```
if condition then  
    statements;  
else  
    alternative-statements;  
END if;
```

PL/pgSQL if-then-else statement

```
if condition then  
    statements;  
else  
    alternative-statements;  
END if;
```

PL/pgSQL if-then-else statement

```
do $$  
declare  
    selected_film films%rowtype;  
    input_film_id films.film_id%type := 100;  
begin  
    select * from films  
    into selected_film  
    where film_id = input_film_id;  
    if not found then  
        raise notice 'The film % could not be found',  
                     input_film_id;  
    else  
        raise notice 'The film title is %', selected_film.title;  
    end if;
```

PL/pgSQL if-then-elsif Statement

- if condition_1 then
- statement_1;
- elsif condition_2 then
- statement_2
- ...
- elsif condition_n then
- statement_n;
- else
- else-statement;
- end if;

PL/pgSQL if-then-elsif Statement

```
do $$  
declare  
    v_film films%rowtype;  
    len_description varchar(100);  
begin  
  
    select * from films  
    into v_film  
    where film_id = 100;  
  
    if not found then  
        raise notice 'Film not found';  
    else
```

PL/pgSQL if-then-elsif Statement

```
if v_film.length >0 and v_film.length <= 50 then
    len_description := 'Short';
elsif v_film.length > 50 and v_film.length < 120 then
    len_description := 'Medium';
elsif v_film.length > 120 then
    len_description := 'Long';
else
    len_description := 'N/A';
end if;

raise notice 'The % film is %.',  

    v_film.title,  

    len_description;
end if;
end $$
```

PL/pgSQL CASE Statement

- The case statement selects a when section to execute from a list of when sections based on a condition.
- The case statement has two forms:
 - Simple case statement
 - Searched case statement

Simple case statement

case search-expression

 when expression_1 [, expression_2, ...] then

 when-statements

 [...]

 [else

 else-statements]

END case;

Searched case statement

- case
- when boolean-expression-1 then
 - statements
- [when boolean-expression-2 then
 - statements
- ...]
- [else
 - statements]
- end case;

While Loop

- do \$\$
- declare
- counter integer := 0;
- begin
- while counter < 5 loop
- raise notice 'Counter %', counter;
- counter := counter + 1;
- end loop;
- end\$\$;

For Loop

- [<<label>>]
- for loop_counter in [reverse] from.. to [by step] loop
 - statements
- end loop [label];

For Loop

- do
- \$\$
- declare
- f record;
- begin
- for f in select title
 from films
- order by title
- limit 10
- loop
- raise notice '%', f.title;
- end loop;
- end;
- \$\$

Introduction to PostgreSQL CREATE PROCEDURE statement

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

Introduction to PostgreSQL CREATE PROCEDURE statement

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

Create Function

- create [or replace] function function_name(param_list)
- returns return_type
- language plpgsql
- as
- \$\$
- declare
- -- variable declaration
- begin
- -- logic
- end;
- \$\$

PL/pgSQL Function Parameter Modes: IN, OUT, INOUT

IN	OUT	INOUT
The default	Explicitly specified	Explicitly specified
Pass a value to function	Return a value from a function	Pass a value to a function and return an updated value.
in parameters act like constants	out parameters act like uninitialized variables	inout parameters act like an initialized variables
Cannot be assigned a value	Must assign a value	Should be assigned a value

Introduction to PL/pgSQL Function Overloading

- PostgreSQL allows multiple functions to share the same name as long as they have different arguments.
- If two or more functions share the same name, the function names are overloaded.
- When you can call an overloading function, PostgreSQL select the best candidate function to execute based on the the function argument list.

How to Develop a PL/pgSQL Function That Returns a Table

- create or replace function function_name (
- parameter_list
-)
- returns table (column_list)
- language plpgsql
- as \$\$
- declare
- -- variable declaration
- begin
- -- body
- end; \$\$

Triggers

- A PostgreSQL trigger is a function invoked automatically whenever an event such as insert, update, or delete occurs.

Triggers

- A PostgreSQL trigger is a function invoked automatically whenever an event associated with a table occurs.
- An event could be any of the following: INSERT, UPDATE, DELETE or TRUNCATE.
- A trigger is a special user-defined function associated with a table.
- To create a new trigger, you define a trigger function first, and then bind this trigger function to a table.
- The difference between a trigger and a user-defined function is that a trigger is automatically invoked when a triggering event occurs.
- PostgreSQL provides two main types of triggers: row and statement-level triggers. The differences between the two kinds are how many times the trigger is invoked and at what time.

Triggers

- For example, if you issue an UPDATE statement that affects 20 rows, the row-level trigger will be invoked 20 times, while the statement level trigger will be invoked 1 time.
- You can specify whether the trigger is invoked before or after an event.
- If the trigger is invoked before an event, it can skip the operation for the current row or even change the row being updated or inserted.
- In case the trigger is invoked after the event, all changes are available to the trigger.

Triggers

- Triggers are useful in case the database is accessed by various applications, and you want to keep the cross-functionality within the database that runs automatically whenever the data of the table is modified.
- For example, if you want to keep the history of data without requiring the application to have logic to check for every event such as INSERT or UPDATE.

Triggers

- You can also use triggers to maintain complex data integrity rules which you cannot implement elsewhere except at the database level.
- For example, when a new row is added into the customer table, other rows must be also created in tables of banks and credits.

Triggers

- The main drawback of using a trigger is that you must know the trigger exists and understand its logic to figure it out the effects when data changes.
- Even though PostgreSQL implements SQL standard, triggers in PostgreSQL has some specific features:
 - PostgreSQL fires trigger for the TRUNCATE event.
 - PostgreSQL allows you to define the statement-level trigger on views.
 - PostgreSQL requires you to define a user-defined function as the action of the trigger, while the SQL standard allows you to use any SQL commands.

PostgreSQL CREATE TRIGGER

- To create a new trigger in PostgreSQL, you follow these steps:
 - First, create a trigger function using CREATE FUNCTION statement.
 - Second, bind the trigger function to a table by using CREATE TRIGGER statement.

PostgreSQL CREATE TRIGGER

```
CREATE FUNCTION trigger_function()
    RETURNS TRIGGER
    LANGUAGE PLPGSQL
AS $$

BEGIN
    -- trigger logic

END;

$$
```

Inheritance

```
CREATE TABLE CUSTOMERS (
    name text,
    age int,
    address text
);
```

```
CREATE TABLE OFFICE_CUSTOMERS(
    office_address text
) INHERITS (customers)
```

Inheritance

```
INSERT INTO CUSTOMERS VALUES('ravi','32','32,  
head street');
```

```
INSERT INTO CUSTOMERS VALUES('michael','35','56,  
gotham street');
```

```
INSERT INTO OFFICE_CUSTOMERS  
VALUES('bane','28','56, circadia street','92 homebush');
```

Non Atomic Columns

```
CREATE TABLE customer (
    name          text,
    Address       text,
    payment_schedule integer[],
);
```

Non Atomic Columns

```
INSERT INTO CUSTOMER_SCHEDULE VALUES( 'jack',
'Athens, Colarado',
'{1,2,3,4}'
)
```

```
INSERT INTO CUSTOMER_SCHEDULE VALUES(
'jackson',
'Tennessey, greece',
'{1,7,3,4}'
)
```

Non Atomic Columns

```
SELECT * FROM CUSTOMER_SCHEDULE WHERE  
CUSTOMER_SCHEDULE.payment_schedule[1] <>  
CUSTOMER_SCHEDULE.payment_schedule[2];
```

Postgres Cluster

- .\initdb.exe -D e:/PostgreSQL/14/data
- Change the server port in data folder config file
- .\pg_ctl.exe start -D e:\PostgreSQL\14\data
- Server started
- psql -h localhost -d postgres -p 5436 -W

Postgres Replication

- #5432
 - CREATE TABLE LogicalReplicationTest (id int PRIMARY KEY);
 - INSERT INTO LogicalReplicationTest VALUES (generate_series(1,10000));
 - SELECT count(*) FROM LogicalReplicationTest;
 - CREATE PUBLICATION testpub FOR TABLE LogicalReplicationTest;
-
- #5436
 - CREATE TABLE LogicalReplicationTest (id int PRIMARY KEY);
 - CREATE SUBSCRIPTION testsub CONNECTION 'host=localhost port=5432 user=postgres password=vignesh dbname=testdb' PUBLICATION testpub;

Basic Transaction Syntax:

```
BEGIN;  
-- SQL operations  
COMMIT; -- Save changes  
-- or  
ROLLBACK; -- Undo changes
```

```
BEGIN;  
  
UPDATE accounts SET balance = balance - 100 WHERE id = 1;  
UPDATE accounts SET balance = balance + 100 WHERE id = 2;  
  
COMMIT;
```

Basic Transaction Syntax:

```
BEGIN;
```

```
UPDATE products SET stock = stock - 5 WHERE id = 100;
```

```
-- Oops: this fails due to constraint
```

```
INSERT INTO logs VALUES (NULL, 'bad insert');
```

```
ROLLBACK; -- Reverts the stock update too
```

Transaction Control Summary

Command	Purpose
BEGIN	Start transaction
COMMIT	Apply changes
ROLLBACK	Undo changes
SAVEPOINT name	Set save point inside tx
ROLLBACK TO SAVEPOINT name	Partial rollback

Postgres Security

-  1. AuthenticationControls who can connect to the database.
- Configured in pg_hba.conf (Host-Based Authentication)
- Common methods:
 - trust: No password (not recommended for production)
 - md5 / scram-sha-256: Password-based (md5 deprecated gradually)
 - peer: OS user must match database user (local Unix)
 - cert: SSL certificate authentication
 - ldap, gss, kerberos, pam, radius: External systems

Postgres Security

- Select name,setting from pg_settings;

name	setting
allow_in_place_tablespaces	off
allow_system_table_mods	off
application_name	pgsql
archive_cleanup_command	(disabled)
archive_command	off
archive_mode	0
archive_timeout	on
array_nulls	60
authentication_timeout	on
autovacuum	0.1
autovacuum_analyze_scale_factor	50
autovacuum_analyze_threshold	2000000000
autovacuum_freeze_max_age	3
autovacuum_max_workers	4000000000
autovacuum_multixact_freeze_max_age	60
autovacuum_naptime	2
autovacuum_vacuum_cost_delay	-1
autovacuum_vacuum_cost_limit	0.2
autovacuum_vacuum_scale_factor	50
autovacuum_vacuum_threshold	-1
autovacuum_work_mem	0
backend_flush_after	safe_encoding
backslash_quote	200
bgwriter_delay	64
bgwriter_flush_after	100
bgwriter_lru_maxpages	2
bgwriter_lru_multiplier	8192
block_size	off
bonjour	hex
bonjour_name	on
bytea_output	0.5
check_function_bodies	32
checkpoint_completion_target	300
checkpoint_flush_after	
checkpoint_timeout	

Postgres Security

-  **2. Authorization (Roles & Privileges)**
- Controls what a user can do.
-  Roles
 - Superusers (CREATE ROLE myuser SUPERUSER)
 - Normal users (LOGIN privilege)
 - Can be granted or revoked:

```
CREATE ROLE readonly;  
GRANT CONNECT ON DATABASE mydb TO readonly;  
GRANT USAGE ON SCHEMA public TO readonly;  
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;
```

Postgres Security

```
CREATE ROLE appuser LOGIN PASSWORD 'secret'  
    NOSUPERUSER NOCREATEDB NOCREATEROLE  
    NOINHERIT NOREPLICATION;
```

Object	Privileges
Table	SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER
Database	CONNECT, TEMP
Schema	USAGE, CREATE
Function	EXECUTE

Postgres Security Object Level Permission

```
GRANT SELECT ON employees TO readonly;  
REVOKE DELETE ON employees FROM appuser;
```

Object	Privileges
Table	SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER
Database	CONNECT, TEMP
Schema	USAGE, CREATE
Function	EXECUTE

Postgres Security Row-Level Security (RLS)

```
ALTER TABLE employees ENABLE ROW LEVEL SECURITY;
CREATE POLICY hr_policy
  ON employees
  USING (department = current_setting('app.current_department'));
ALTER TABLE employees ENABLE POLICY hr_policy;
```

Use SET app.current_department = 'HR'; at session level.

Questions



Module Summary

- In this module we discussed
 - Installing Postgres
 - Querying Postgres
 - Data Management
 - PL/SQL

