Member-only story

SQL Query Journey in Postgres

Oz · Following
5 min read · Dec 26, 2024

Listen Share More

In PostgreSQL, when a client sends an SQL query, it undergoes a complex journey from submission to execution and result retrieval. This journey involves multiple processes, from the Postmaster managing connections to the backend processes handling parsing, planning, and execution. Along the way, PostgreSQL ensures data consistency, durability through Write-Ahead Logging (WAL), and efficient data retrieval using shared buffers and disk storage. Understanding this flow helps optimize performance and ensures smooth interaction between the client and the database.

1. The Client's Request

The story begins with the **client** (for example, `psql` , a web application, or any database-connected client) sending an SQL query to the PostgreSQL server.

- The client typically connects to the PostgreSQL server using the **TCP/IP protocol** (or a Unix socket, depending on the configuration). The client sends an **SQL query;** `SELECT * FROM users WHERE age > 30`

2. Connection Establishment and the Postmaster Process

The **Postmaster process** is always running as the central management process in PostgreSQL. Here's what happens:

- When the query arrives at the server, the **Postmaster** process listens on the designated port (5432 by default).

- The **Postmaster** accepts the connection request and forks a **child process** (a **backend process**) dedicated to the client. Each client gets its own backend process to ensure isolation.
- The backend process manages the entire lifecycle of the query: from parsing the SQL statement, planning its execution, performing the actual query execution, to finally sending the result back to the client.

At this point, the **client-server connection** is established, and the backend process is ready to execute the query.

3. Query Parsing and Planning

Once the backend process has been forked, the query execution journey begins. The backend process performs several key tasks:

Parsing:

- The query is first **parsed** to ensure it is syntactically correct. The backend process checks if the SQL statement adheres to PostgreSQL's grammar and can be processed.
- **Errors** in the query (e.g., missing semicolons, wrong column names, or incorrect SQL syntax) are detected here, and the backend returns an error to the client if needed.

Planning:

- After parsing, PostgreSQL's **query planner** takes over. The planner is responsible for determining the **best execution plan** for the query. This involves selecting indexes (if any), joining tables in an efficient way, and choosing the most optimal strategy for data retrieval.
- PostgreSQL uses various strategies, such as **sequential scan**, **index scan**, or **hash join**, depending on the query and the available data.

4. Execution: The Data Retrieval Process

At this stage, the query planner has decided the most efficient way to retrieve data. The backend process now executes the plan. Here's what happens during execution:

Accessing Memory (Shared Buffers):

- Before going to disk, PostgreSQL will **check the shared buffers** to see if the requested data is already in memory. This is important for performance since reading from memory is much faster than reading from disk.
- **Shared Buffers** hold the database pages (usually 8 KB each) that have been read from disk or modified by previous transactions.
- If the data is already in memory (a **cache hit**), the backend can proceed to the next step without hitting the disk.

Disk Access (When Data is Not in Memory):

- If the requested data is not in the shared buffers (**cache miss**), PostgreSQL retrieves it from disk.
- The disk data is first read into the **shared buffers**, and then the backend process continues with executing the query.
- In this process, PostgreSQL reads data from the **data files** that store the actual table content. It handles locking, consistency, and data retrieval in a manner that ensures no conflicting operations are occurring simultaneously (thanks to **MVCC** — Multi-Version Concurrency Control).

5. Write-Ahead Logging (WAL): Ensuring Durability

PostgreSQL's durability and crash recovery rely heavily on **Write-Ahead Logging (WAL)**. Here's how it works during query execution:

- **Before any changes to data** are made (in the case of an `UPDATE` or `INSERT` query, for example), PostgreSQL **writes the change to the WAL**. This ensures that the database's state can be recovered even if the system crashes after the data is modified but before it's written to the actual data files.
- The **WAL writer process** runs in the background, continuously writing changes from the shared buffers to the WAL files, ensuring durability.
- WAL logs are written before any changes occur to the actual data files (the **data directory**), making sure that PostgreSQL can recover any changes made during a transaction.

6. Data Modification and the Checkpoint Process

For **data-modifying queries** (like `INSERT` , `UPDATE` , `DELETE`), the backend process performs the following:

- Once the WAL is written, the query changes the data in memory (shared buffers).
- **Checkpoints** occur periodically in PostgreSQL, where the system flushes all modified pages from the shared buffers to the actual **data files** on disk. This ensures that all changes made since the last checkpoint are saved to disk.
- If the server crashes, the WAL entries allow PostgreSQL to recover any changes that happened after the last checkpoint, bringing the system back to a consistent state.

7. Query Results: Returning Data to the Client

Once the backend process has completed executing the query, it prepares the results for the client. Here's the final step:

Preparing the Response:

- If the query was a **SELECT**, the backend process retrieves the relevant rows of data (either from memory or from disk) and packages them into a result set.
- For data-modifying queries, the backend may return an acknowledgment, such as the number of rows affected (e.g., `INSERT 0 1` for a single insertion).

Sending the Response:

- The results are sent from the backend process to the client through the established connection.
- If the query was executed successfully, the results are transmitted back to the client (e.g., in the form of rows and columns for a `SELECT` query, or a success message for a `INSERT`).
- The client then receives the response and processes it according to its logic.

8. Cleanup and Process Termination

- After the query is executed and the results are sent to the client, the backend process performs **cleanup** operations. This includes:
- Closing open file descriptors.
- Releasing locks on tables and rows.
- Freeing up memory used by the query execution.

- If the client disconnects, the backend process terminates. The Postmaster process continues to manage other incoming connections.

In Summary: The Complete Journey of an SQL Query in PostgreSQL

1. **Client sends the SQL query** to the PostgreSQL server.
2. **Postmaster process** listens for incoming connections and forks a **backend process** for the client.
3. The backend process **parses** and **plans** the query, optimizing it for execution.
4. **Memory (shared buffers)** is checked for the data. If not found, PostgreSQL fetches it from disk.
5. **Write-Ahead Logging (WAL)** ensures the durability of data changes before modifications are made.
6. The backend process executes the query, **modifies data** if necessary, and performs a **checkpoint**.
7. **Results** are returned to the client, or the number of rows affected is acknowledged.
8. After sending the result, the backend process **cleans up** and terminates.

This journey, from query submission to result return, illustrates the seamless interaction between the client, PostgreSQL processes, and the disk, ensuring efficiency, durability, and data integrity throughout the entire cycle.

Postgresql Acid Mvcc

Following

Written by Oz

149 Followers · 13 Following

Database Administrator 🐘

No responses yet

Gvadakte

What are your thoughts?

More from Oz

Queries

time	Service	username	Query	Execution_time
2024-09-26 13:15:00	pmm-server-postgresql	pmm-ma…	SELECT /* agent='pgstatstatements' */ "pg_stat_statements"."userid", "pg_stat_statements"."dbid", "pg_stat_statements"."queryi…	0.0242
2024-09-26 13:19:00	pmm-server-postgresql	pmm-ma…	SELECT current_database() datname, schemaname, relname, heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit, toast_b…	0.0165
2024-09-26 13:10:48	pmm-server-postgresql	pmm-ma…	SELECT * FROM pg_stat_database_conflicts	0.0159
2024-09-26 13:18:00	pmm-server-postgresql	pmm-ma…	SELECT /* agent='pgstatstatements' */ "pg_stat_statements"."userid", "pg_stat_statements"."dbid", "pg_stat_statements"."queryi…	0.0131
2024-09-26 13:19:00	pmm-server-postgresql	pmm-ma…	SELECT /* agent='pgstatstatements' */ "pg_stat_statements"."userid", "pg_stat_statements"."dbid", "pg_stat_statements"."queryi…	0.0131
2024-09-26 13:12:00	pmm-server-postgresql	pmm-ma…	SELECT /* agent='pgstatstatements' */ "pg_stat_statements"."userid", "pg_stat_statements"."dbid", "pg_stat_statements"."queryi…	0.0126
2024-09-26 13:15:00	pmm-server-postgresql	pmm-ma…	SELECT current_database() datname, schemaname, relname, heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit, toast_b…	0.0113

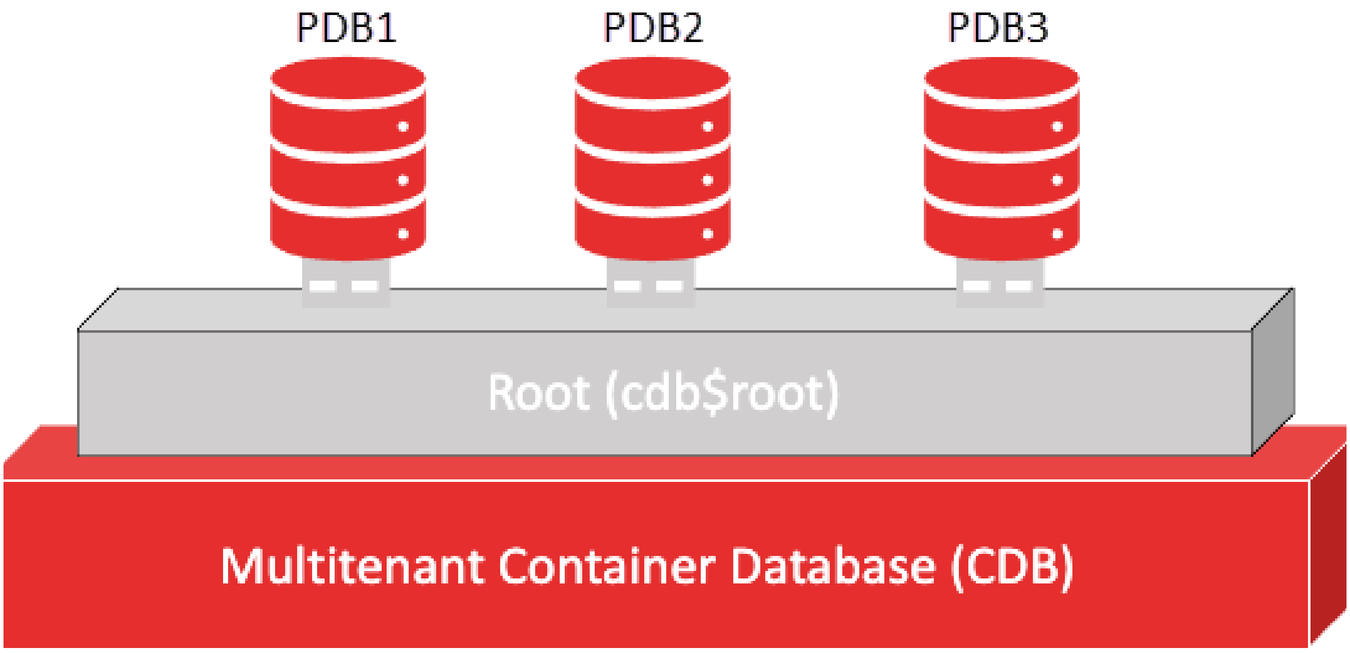
1 - 7 of 538 rows

Oz

Installing Percona Monitoring & Management (PMM) with Postgres

Introduction:

Sep 26, 2024 54 1



Oz

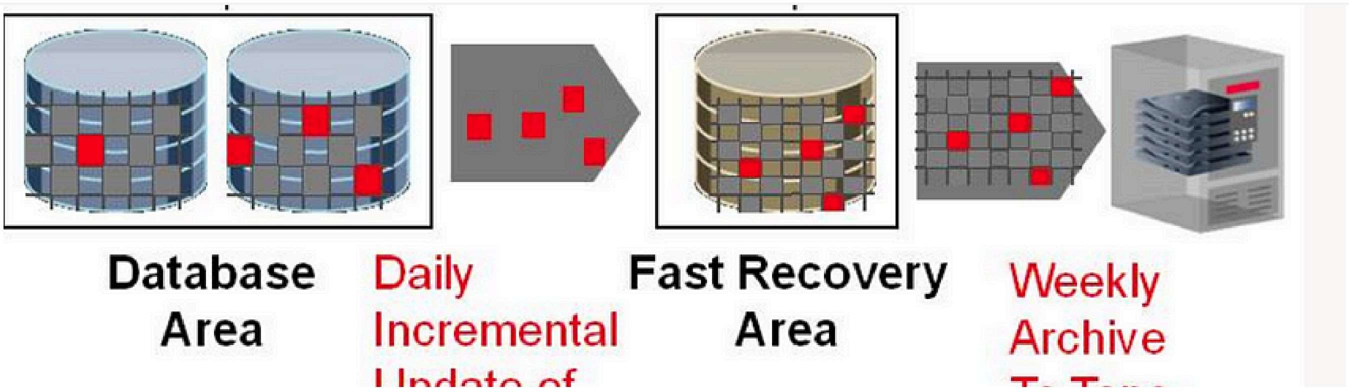
Pluggable Database Command

—————————— -————————————— - create pluggable database pdb1 admin user root identified by test123; alter pluggable database…

May 12, 2023

Open in app ↗

Medium Search



Oz

RMAN Backup Basic Commands

rman target / rman target sys/password@YDKTST; backup database; backup database format '/backup/path/%d_%t_%s.rman'; backup tablespace…

May 11, 2023 1

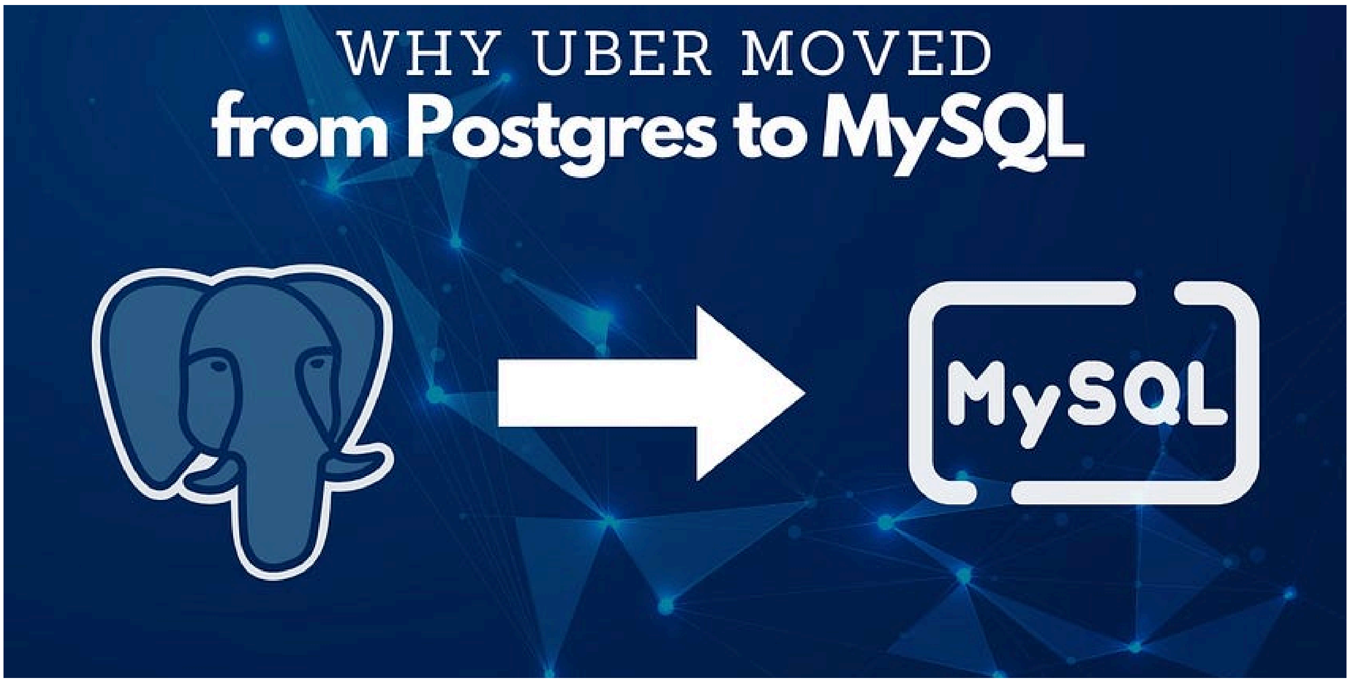
Oz

delete jobs

May 8, 2023

See all from Oz

Recommended from Medium



In Databases by Sergey Egorenkov

Why Uber Moved from Postgres to MySQL

How PostgreSQL's architecture clashed with Uber's scale—and why MySQL offered a better path forward

Mar 29 228 7

Tihomir Manushev

Vector Search with pgvector in PostgreSQL

Simple AI-powered similarity search

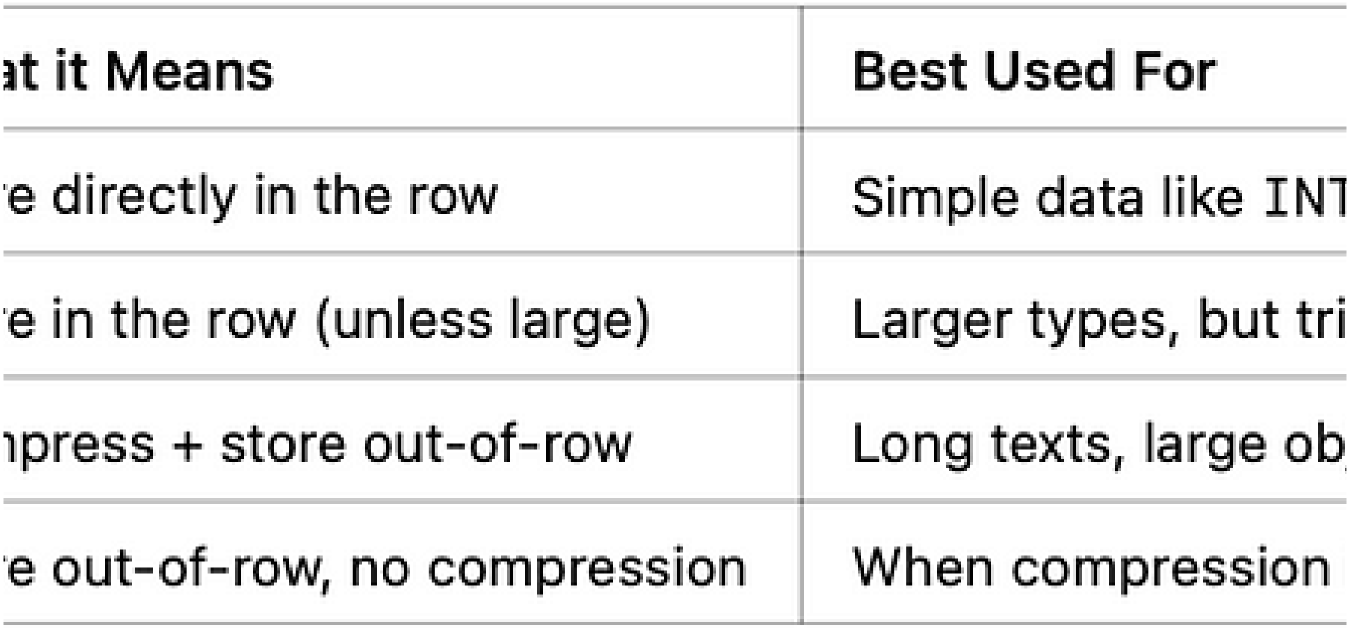
Mar 9

Vedant Jadhav

Understanding User, Roles and Privileges in PostgreSQL

Recently, I found myself in a situation where I had to grant specific database privileges and access to my colleagues based on their job...

Feb 3 1 1



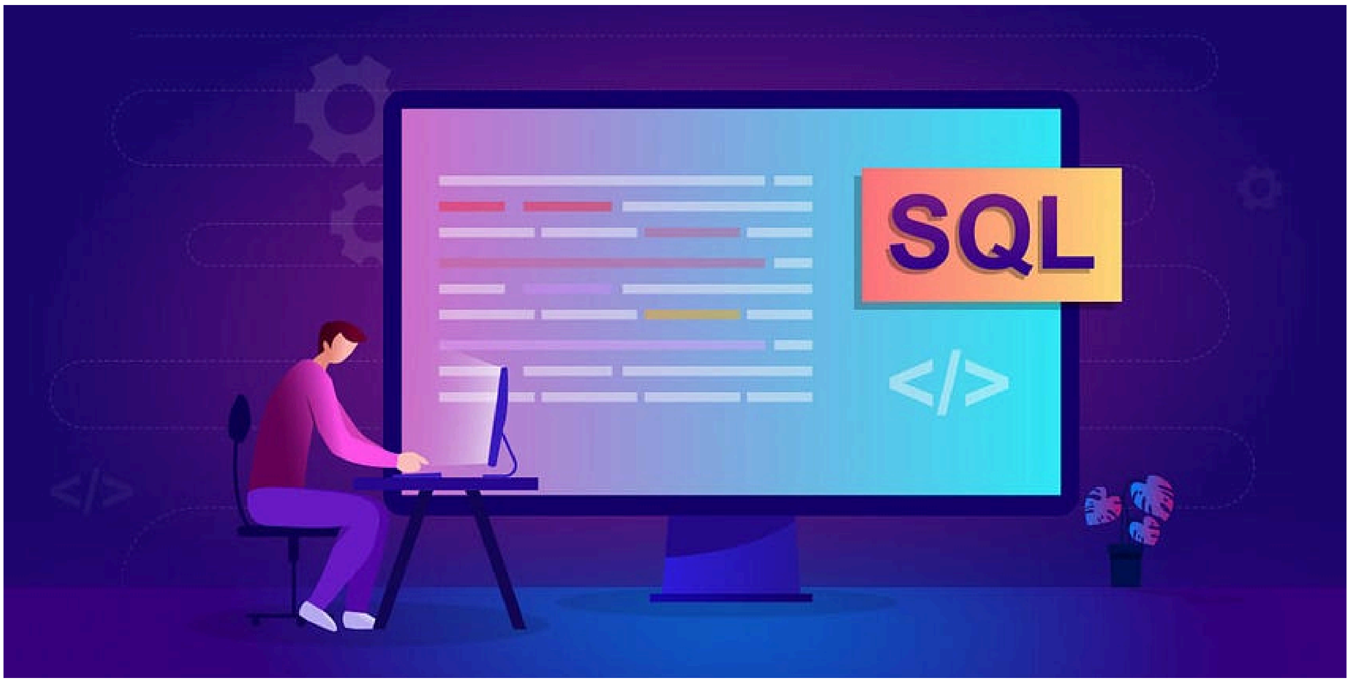
at it Means	Best Used For
e directly in the row	Simple data like INT
e in the row (unless large)	Larger types, but tri
npress + store out-of-row	Long texts, large ob
e out-of-row, no compression	When compression

Udbhav Singh

Storages in PostgreSQL

What Are Storage Types in PostgreSQL—And Why Should You Care?

6d ago 18



In Hack the Stack by Coders Stop

9 Database Optimization Tricks SQL Experts Are Hiding From You

Most developers learn enough SQL to get by—SELECT, INSERT, UPDATE, DELETE, and maybe a few JOINs. They might even know how to create…

Mar 27 185 5

Ajaymaurya

How Often Should You Reindex Your PostgreSQL Database? A Data-Driven Approach

PostgreSQL is one of the most powerful and flexible relational databases available today. However, like any database system, it requires…

Mar 24

See more recommendations