PostgreSQL Internals Part 3: Understanding Processes in PostgreSQL

PostgreSQL Background Processes (High-level)

□□□□□□□ PostgreSQL instance □□□□ □□□□, □□□□□□□ □□□□ mandatory background processes □□□□ □□□□□□.
□□□□□□□□ □□□ database □□ health, performance □□□ reliability maintain □□□□ □□□□.

□□□□□□ processes:

Postgres Server Process → client connections handle □□□□ (□□□□□□□□□ client □□□□ □□□ process fork □□□□).

Background Writer → dirty pages □□□□□□□ disk □□ flush □□□□.

Checkpointer → checkpoint record □□□□□□ □□□□□□□ □□□□□□□□□□ dirty pages flush □□□□.

WAL Writer → WAL (Write Ahead Log) buffer disk □□ □□□□□□□.

Autovacuum Launcher + Workers → dead tuples □□□ □□□□□□ (vacuum).

Archiver → WAL logs archive □□□□ (backup, PITR □□□□).

Stats Collector / PgStat → database activity statistics □□□ □□□□.

Logical Replication Launcher → replication jobs □□□□□□□ (□□ configure □□□□ □□).

Syslogger → logs manage □□□□.

? □□□ "dirty pages" context □□□□□□ □□ processes □□□ □□□ □□□□□□:
1. WAL Writer

□□□□□□□ tuple update/delete □□□□ → change □□□□□□ WAL buffer □□□□□□ □□□□.

WAL writer □□ buffer □□□ disk □□ (pg_wal directory □□□□□□) flush □□□□.
? □□ data safety □□□□: crash □□□□ □□□ WAL □□□□ recovery □□□□ □□□□.

2. Background Writer

□□□□□□□ □□□ RAM □□□□□□ (shared buffer □□□□□□) □□□□ → □□□□ page □□ "dirty" flag □□□□□□.

Background writer □□ 200ms □□ wake □□□□ □□□□ dirty pages disk □□ flush □□□□.
? □□ □□□□□: checkpoint □□□□ □□□□□ □□□ load □□□ □□□.

3. Checkpointer

□□□□□□□□□ checkpoint □□□□:

WAL □□□□□□ checkpoint record □□□□□□□ → “□□ □□□□□□□□□□□□□□ □□□ □□□□□□□□□ □□□□ □□□□”.

□□ □□□□ □□□□□□□□□□□□□□ □□□□ dirty pages disk □□ □□□□□□ □□□□□□.
? □□□□□□ □□□□□ writing □□□□ (□□ background writer □□ □□□□□□ □□□ □□□□□ □□□□□ □□).

4. Autovacuum Workers

□□□□□□□ tuples delete □□□□□□ update □□□□□□ → □□□□□ tuples "dead" □□□□□□ (□□ disk □□ □□□□□ □□□□□□).

Autovacuum process □□ dead tuples □□□□□□ □□□□□□.

□□□□□□□ bloat □□□ □□□□ □□□ future queries fast □□□□□□□.

□ Example:

□□ autovacuum □□□ □□□□ → millions of dead tuples □□□ □□□□□ → queries slow → disk usage □□□□□.

5. Archiver

□□ WAL archiving enabled □□□□ (archive_mode = on) → WAL files archive □□□□ safe location □□ □□□□□□□ (backup, DR □□□□□).

6. Stats Collector

□□□□□□□□□ query execution, table usage, tuple insert/update/delete □□ □□□□□□□ □□□□□ □□□□□.

Autovacuum decisions, query planner cost calculations □□□□ statistics □□□□□□□ □□□□□□.

□ Example Case — □□□□ processes □□□□□□ □□□ □□□ □□□□□□?

□□□□ □□ banking application □□□□□□ transactions □□□□□□□ heavy load □□□:

User transaction insert □□□□ →

Change RAM (shared buffer) □□□□□□ dirty page □□□□□□□ □□□□□.

WAL record □□□□ □□□□□ (WAL writer disk □□ □□□□□□□).

Background writer → □□□ □□□□□ dirty pages □□□□□□□□ □□□□□□.

Checkpoint □□□□ → □□□□□□□ dirty pages □□□□□ □□□□□□□□□ □□□□□□□ □□□□□□.

Delete / Update □□□□ □□□□□□□ tuples dead □□□□ →

Autovacuum worker □□□□□□ → dead tuples □□□ □□□□□.

Archiver → checkpoint □□□□□□□ WAL logs archive □□□□□□ □□□□□□□.

Stats collector → □□□□ inserts/deletes □□□□□, □□□□ dead tuples □□□□□, □□ data track □□□□.

□ Marathi Analogy (□□□□□□□ □□□□□□□)

WAL Writer = □□□□□□□ notebook □□□□□□ □□□□□ notes □□□□□□ □□□□□□ (□□□□□ □□□□□□□□ □□□ safe).

Background Writer = □□□ □□□□□ □□□□ papers cupboard □□□□□□ □□□□□□□□ clerk.

Checkpointer = □□□□□□□□□□□□ □□□□□□ □□□ audit □□□□ □□□ □□□□□□□ "□□□□ □□□□ cupboard □□□□□□ □□□□□".

Autovacuum = □□□□□ □□□□□□□ → □□□□□, □□□□□□ papers □□□□□□ □□□□□□.

Archiver = □□□□□□ □□ → □□□□□□□□□ papers photocopy □□□□ □□□□□□□□ safe cupboard □□□□□□ □□□□□.

Stats Collector = attendance register → □□□ □□□□ □□□ □□□□□□ □□□□□ □□□□□ □□□□□□.

□ □□□□□□□

Checkpointer + Background Writer → dirty pages manage □□□□□□.

WAL Writer + Archiver → data durability □□□ recovery handle □□□□□□.

Autovacuum → dead tuples □□□ □□□□□ □□□□ □□□□□□□ □□□□□□.

Stats Collector → database □□ self-tuning □□□□ information □□□□.
Updated: 14 May, 2024.
Welcome to the third part of our PostgreSQL internals series. In this blog, we'll explore the concept of processes, their architecture, the different types of processes available, and their respective responsibilities.

If you're interested in exploring the earlier parts of this series where we delved into database clusters, databases, and tables, you can find them here:

- Part 1: Understanding database cluster, database and tables
- Part 2: Understanding Page Structure

PostgreSQL is a relational database management system based on a client and server architecture. Within this model, a process represents an individual program instance running on the server, responsible for specific tasks like client connections, query execution, or background maintenance. These processes collaborate to oversee the database and address user inquiries.

Every operation in PostgreSQL is handled by a separate process, with each action following an append-only approach. This means that whenever you insert, update, or delete a record, PostgreSQL doesn't alter existing data directly. Instead, it creates new data or marks existing data as obsolete, maintaining a historical record of changes. This append-only method not only ensures data integrity but also facilitates efficient database management over time, providing a robust system for tracking modifications without the risk of directly overwriting valuable information.

We have the following types of Processes in PostgreSQL

Postgres server process

This is the main process that manages everything in PostgreSQL. For now, let's assume it is a parent process of everything that runs in PostgreSQL It creates a listener on the configured interfaces and port (default 5432). It is also responsible for forking other processes to do various tasks. In the older version of PostgreSQL, we used to call it as Postmaster process. When initiating the pg_ctl utility with the start option, it triggers the launch of a Postgres server process. This process initializes a shared memory area in memory (elaborated below) and initiates various background processes, including those related to replication and background workers as required. Subsequently, it awaits client connection requests. Upon receiving such a request, it initiates a backend process to handle the client's tasks. Each connection request from a client would get a dedicated backend process (The started backend process then handles all queries issued by the connected client.)

NOTE: A Postgres server process listens on a designated network port, typically set as the default port 5432. While multiple PostgreSQL servers can operate on the same host, each server must be configured to listen on distinct port numbers, such as 5432, 5433, and beyond.

Shared memory

All the processes in PostgreSQL need to talk to each other so we have a shared memory area that plays a crucial role in enabling efficient communication and data sharing between backend processes within the PostgreSQL server.
When you start the PostgreSQL server using pg_ctl start, the shared memory segment in RAM gets allocated by the PostgreSQL server process. This segment is a special memory region that can be accessed and used by all PostgreSQL backend processes simultaneously. Here are some key use cases for this
A significant portion of the shared memory area is used for PostgreSQL's shared buffers. This cache stores frequently accessed data blocks from tables and indexes. When a backend process needs to access data, it first checks the shared buffers. If the data is present (cache hit), it can be retrieved much faster than reading it from disk. This significantly improves query performance.
In shared memory, WAL and pages record can live as well or everything that needs to be shared between the processes lives here it can be increased by the shared_buffer parameter inside the postgresql.conf file A larger shared memory area can improve query performance by allowing the shared_buffers cache to hold more frequently accessed data blocks. This reduces disk I/O and speeds up data retrieval.

backend process

It is also called a Postgres process and it is usually started by the main process(postgres server process) Since a backend process is only allowed to operate on one database, you must explicitly specify the database you want to use when connecting to a PostgreSQL server. Every client that connects with the PostgreSQL server gets its dedicated backend process. Clients are not ones who directly connect with psql but they are usually an application that is trying to connect with the PostgreSQL server so it's rare that someone tries to connect with PostgreSQL directly with psql. Backend process handles all queries and statements issued by a connected client. How many clients can connect to the database simultaneously depends on the value of the max_connections property available inside postgresql.conf file by default its value is 100 means only 100 clients can connect to the server at a time
If many clients, such as WEB applications, frequently connect and disconnect from a PostgreSQL server, it can increase the cost of establishing connections and creating backend processes

NOTE: PostgreSQL does not have a native connection pooling feature. Hence some connection pooler is recommended to use here i.e, **pgpool** or **pgbouncer**

background processes

Background processes in PostgreSQL are system-level processes that perform various administrative tasks and housekeeping activities to ensure the smooth operation and integrity of the database system. These processes run independently of client connections and backend processes, continuously monitoring and managing different aspects of the database server. Examples of these processes could be **VACUUM** and **CHECKPOINT**

Background writer

Its job is to flush the dirty pages to the file system or persistent location from shared memory. The operating system employs an in-memory file system cache to collect write operations until a sufficient amount has accumulated. It then consolidates and writes these changes to disk collectively to minimize disk I/O. However, The background writer function serves to alleviate memory pressure by regularly flushing these dirty pages from memory to the file system, thus making space for additional pages to enter the buffer.
You might wonder, isn't it risky to save changes in memory? What if the database crashes? Wouldn't we lose everything? Actually, no. We also save the changes to something called Write-Ahead Logging (WAL) and store it on disk frequently, especially when a transaction is committed. So, even if a crash happens, we can retrieve what's on disk and replay the changes from WAL to the data pages to reach the final state.

Checkpointer

While explaining the background writer we mentioned that the background writer transfers the unsaved pages from shared buffers to the file system to make room in shared buffers. Although changes eventually reach the disk, they temporarily reside in the operating system's file cache. This delay allows the operating system to combine multiple writes into a single I/O operation for efficiency. However, there's a risk of losing pages in the file system cache if there's a crash, so databases don't rely on it for durability.
To overcome this issue we have another background process called Checkpointer, which ensures that pages are written directly to disk without going through the file system cache. It also creates a checkpoint record to ensure that the Write-Ahead Log (WAL) and data files are fully synchronized. In the event of a crash, this checkpoint serves as the starting point for redoing changes from the WAL, which has also been written to disk.

Redo

In PostgreSQL, redo refers to the process of reapplying changes to the database that were previously recorded in the Write-Ahead Logging (WAL) files. When a transaction is committed, its changes are first written to the WAL before being applied to the actual data files. In the event of a crash or failure, PostgreSQL uses the WAL to redo these changes during crash recovery, ensuring that the database is brought back to a consistent state. Redoing changes from the WAL ensures durability and maintains data integrity, even in the face of system failures.

Auto vacuum launcher

This process periodically invokes the autovacuum worker processes for the vacuum process. In PostgreSQL, a vacuum process is a background operation that serves several important functions for maintaining the health and efficiency of the database. Its primary purpose is to reclaim storage space and improve performance by removing obsolete or dead tuples (rows) and updating internal data structures. So this Auto Vacuum Worker process can be used for

- Dead Tuple Removal
- Update Statistics
- Clean Up Indexes

WAL writer

PostgreSQL uses a Write-ahead log (WAL) to record changes made to its data for durability. The WAL is stored as records in shared memory. When transactions happen, many processes write to the WAL. But eventually, the WAL needs to be saved in files on disk, not just in the file system cache. It needs to be physically on disk to be helpful. So WAL Writer process is used to flush the WAL data available on the WAL buffer to persistent storage or on the disk

WAL senders and WAL receivers

WAL senders are background processes running on the primary server in a PostgreSQL replication setup. They continuously send Write-Ahead Log (WAL) data from the primary server to one or more standby (replica) servers. These processes transmit WAL data over the network to keep standby servers updated with changes made on the primary server. By replicating WAL data, WAL senders enable standby servers to maintain an identical copy of the primary database, ensuring data redundancy and fault tolerance.

The WAL receiver is a background process running on standby servers in a PostgreSQL replication setup. It receives WAL data sent by the primary server's WAL sender processes. The WAL receiver applies the received WAL data to the standby server's local database, ensuring that the standby server stays synchronized with the primary server's state.

WAL archiver

As transactions take place in the database, PostgreSQL generates corresponding WAL records, storing them in designated WAL files within the pg_wal directory. When a WAL segment is full or no longer required for crash recovery, it becomes a completed WAL segment. The decision on whether a WAL segment is still necessary is determined by the checkpoint process.
After the checkpointer process establishes a checkpoint, older WAL records can be safely discarded. PostgreSQL archives WAL entries. This task is managed by the WAL archiver, which regularly scans the server's WAL directory for completed segments. Upon detection, the archiver copies these segments to a specified archival location.

Logging collector

This process writes all the log messages including simple outputs and errors into log files.

Startup process

This is the first process that gets started when we start the PostgreSQL server with the pg_ctl start command. During our discussion about the background writer and checkpointer, we mentioned that if a crash occurs, PostgreSQL relies on Write-Ahead Logging (WAL) to restore the data. The startup process handles this task to perform recovery and no other activities can proceed until the startup process is completed. Thus no connections are allowed until the database is fully recovered.

Replication-associated processes

This process is used to perform streaming replication. Streaming replication involves two types of processes working together. A walsender process on the primary server sends Write-Ahead Logging (WAL) data to the standby server. Subsequently, a walreceiver and a startup process on the standby server receive and replay this data. Communication between a walsender and a walreceiver occurs via a single TCP connection.

Background worker processes

Background worker processes in PostgreSQL enable the execution of user-defined processes, managed by the server, with their lifecycle tied to the server's status. These processes operate within PostgreSQL's shared memory area, allowing them to connect to databases internally and execute transactions serially, akin to regular client-connected server processes. They can be initialized during PostgreSQL startup by configuring the shared_preload_libraries parameter.