## Introduction

PostgreSQL, also known as “Postgres,” is a popular open-source database management system. It is known for its strong support for reliability, data integrity, and concurrency. Postgres has a large and active development community and is widely used in businesses, government agencies, and other organizations around the world.

Its path in the industry as open-source software is the result of the initiative andefforts of many people through the years.

## PostgreSQL release cycle

As shown in the graph (*Figure 1.3*), the first version of PostgreSQL was released in 1996, and in 2022, PostgreSQL 15 version will be released:.*(Reference: Versioning)*

Please find the *Table* containing release dates for each major version:

More information <https://www.postgresql.org/support/versioning/>

**Better Handling of Massive Amounts of Data**

The following table describes various hard limits of PostgreSQL. However, practical limits, such as performance limitations or available disk space may apply before absolute hard limits are reached

**Table K.1. PostgreSQL Limitations**

| **Item** | **Upper Limit** | **Comment** |
| --- | --- | --- |
| database size | unlimited |  |
| number of databases | 4,294,950,911 |  |
| relations per database | 1,431,650,303 |  |
| relation size | 32 TB | with the default BLCKSZ of 8192 bytes |
| rows per table | limited by the number of tuples that can fit onto 4,294,967,295 pages |  |
| columns per table | 1,600 | further limited by tuple size fitting on a single page; see note below |
| columns in a result set | 1,664 |  |
| field size | 1 GB |  |
| indexes per table | unlimited | constrained by maximum relations per database |
| columns per index | 32 | can be increased by recompiling PostgreSQL |
| partition keys | 32 | can be increased by recompiling PostgreSQL |
| identifier length | 63 bytes | can be increased by recompiling PostgreSQL |
| function arguments | 100 | can be increased by recompiling PostgreSQL |
| query parameters | 65,535 |  |

**How a connection is established in PostgreSQL**

## PostgreSQL is implemented using a simple “process per user” client/server model. In this model, every client process is connected to exactly one server process.

**Step by Step:**

* Client process requests the master server process called POSTMASTER process for a connection to the database instance.
* The postmaster process creates a new backend process called Postgres process, after authentication check is done and if everything is fine, the connection is created.
* The postmaster process will then assign the newly created backend process to the client process.
* Once a connection is established the client process can send a query to the *backend* (server). The query is transmitted using plain text, i.e., there is no parsing done in the *frontend* (client).
* The server parses the query, creates an *execution plan*, executes the plan and returns the retrieved rows to the client by transmitting them over the established connection

**PostgreSQL Connection string**

You need to specify the following parameters to connect to PostgreSQL:

* Host or host address
* Port
* Database name
* User
* Password (or other means of authentication, if any)

**test\_db=> SELECT version();**

version

----------------------------------------------------------------------------------------------------------

PostgreSQL 10.11 on x86\_64-pc-linux-gnu, compiled by gcc (GCC) 4.4.7 20120313 (Red Hat 4.4.7-23), 64-bit

(1 row)

**Examples notation**

| **Notation** | **Description** |
| --- | --- |
| # | shell prompt for Linux root user |
| $ | shell prompt for Linux general user |
| postgres=# | psql prompt for PostgreSQL administrator |
| postgres=> | psql prompt for PostgreSQL general user |

**Help commands**

**postgres=# \?**

* Example.,

\ds[S+] [PATTERN] list sequences

\dt[S+] [PATTERN] list tables

\du[S+] [PATTERN] list roles

* \h

**postgres=# \h**

Available help:

ABORT CREATE FOREIGN TABLE

ALTER AGGREGATE CREATE FUNCTION DROP SEQUENCE

…

…

**postgres=# \h ALTER DATABASE**

Command: ALTER DATABASE

Description: change a database

Syntax:

ALTER DATABASE name [ [ WITH ] option [ ... ] ]

where option can be:

ALLOW\_CONNECTIONS allowconn

CONNECTION LIMIT connlimit

IS\_TEMPLATE istemplate

ALTER DATABASE name RENAME TO new\_name

ALTER DATABASE name OWNER TO { new\_owner | CURRENT\_USER | SESSION\_USER }

ALTER DATABASE name SET TABLESPACE new\_tablespace

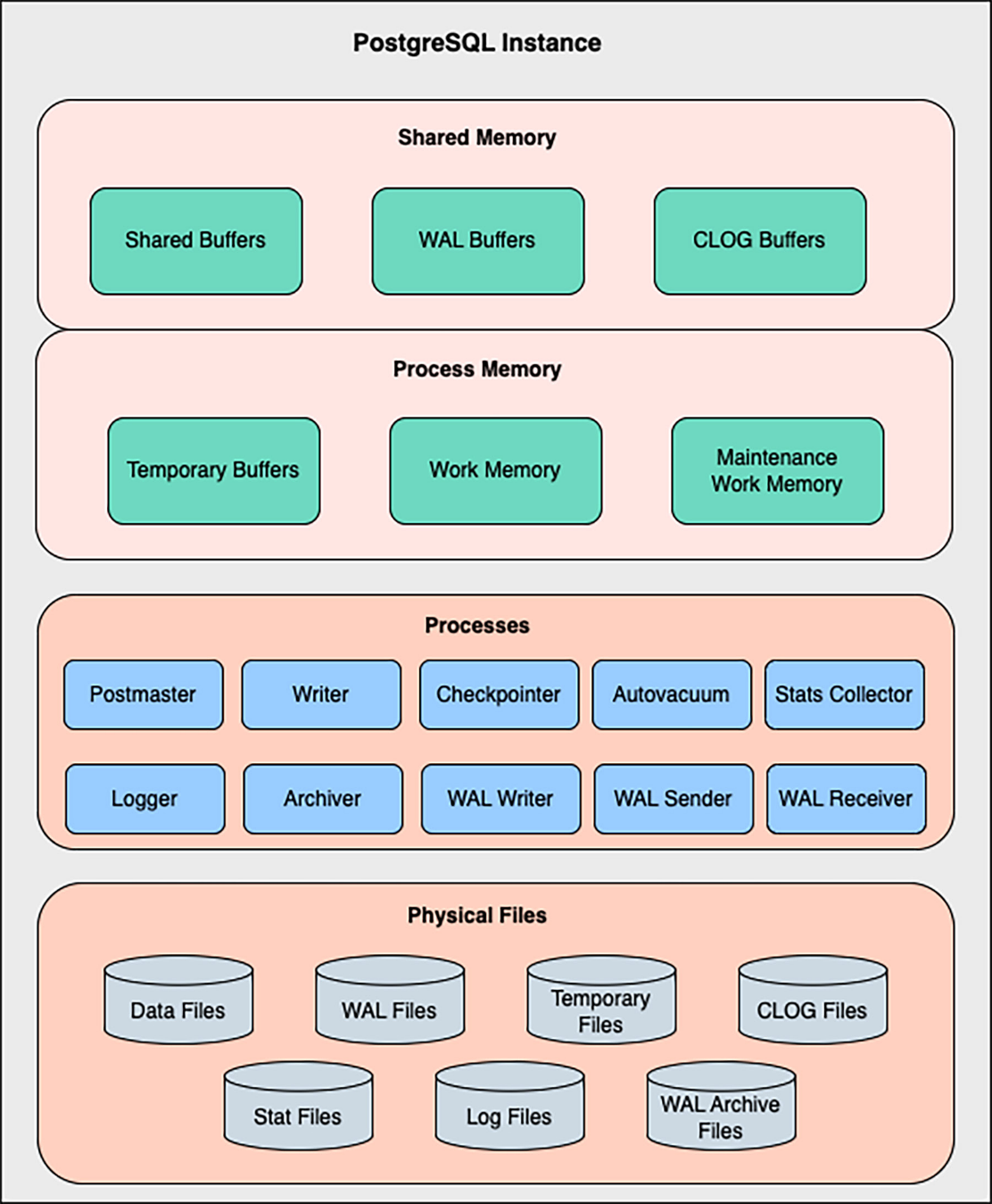
ALTER DATABASE name SET configuration\_parameter { TO | = } { value | DEFAULT }

ALTER DATABASE name SET configuration\_parameter FROM CURRENT

ALTER DATABASE name RESET configuration\_parameter

ALTER DATABASE name RESET ALL

**PostgreSQL System Architecture**



## Introduction

Like other modern **Relational Database Management Systems** (**RDBMS**), PostgreSQL results from multiple components existing on top of an operating system layer. Each one is responsible for some specific aspects of the whole functionality. Understanding these different parts will prepare you for any technical interview or for debugging and resolving issues on your running systems.

## Structure

In this chapter, we will learn about the three main components of the PostgreSQL architecture and review some details about their internal functionality.

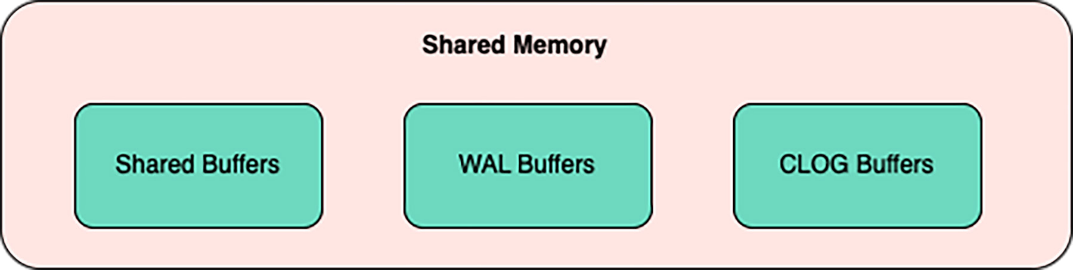
* Memory architecture
* Background processes
* Physical file

**Objectives**

In the content of this chapter, we will review the design of the PostgreSQL architecture and how its different components work and interact. Every sub-section will focus on one main component and explain what is involved and how it works. At the end of the chapter, you will become familiar with the various memory areas, the processes responsible for the different tasks, and the physical file layout.

## Memory architecture

We could say in a simplified way the purpose of PostgreSQL (and of any other RDBMS) *is to retrieve the queried data as fast as possible*. To achieve that goal, the Database system attempts to operate most of the data access and manipulations (READS and WRITES) directly in memory (RAM). Reading and writing data in memory is much faster than in any other media, such as disks. Shared memory

The **shared memory** is allocated when the PostgreSQL instance is started and divided into multiple fixed-size sub-areas. *Figure 5.1* shows the sub-areas:

***Figure 5.1:*** *Shared Memory areas*

### Shared buffers

We might say the Shared Buffers sub-area is one of the most important since it is where the database data will be written and read as most as possible. In other Database technologies, the equivalent to this area is also known as the *database cache*. When a row (tuple in the postgres argot) is read the first time, it needs to be retrieved from disk, which is an expensive and slower task, but then the row is stored in this memory area, so any new query that might require it can get it quickly from here without going to the disk. Also, every time a row is added or modified, the writes are done here.

When a data page (or data block), which is the most elemental data unit (equivalent to 8KB for default), changes its contents in the Shared Buffer, then is marked as a “dirty page”. Subsequently, the dirty data is flushed to disk by some of the background processes, and it is persisted in the physical files called “data files”. We will get a deeper look at these later in this chapter.

At the configuration level, the area of the shared buffer is controlled by the parameter **shared\_buffers**. Usually, 25% of the total RAM is set as the size of the shared buffer. Modifying this parameter required a PostgreSQL restart.

### WAL buffers

The **Write Ahead Log** (**WAL**) is a transaction log mechanism used in PostgreSQL to keep a log of all the metadata related to the data changes. Hence, it is possible to redo the data operations in the case of data recovery.

Every time a COMMIT is executed, the changed data is stored in the Shared Buffers, as we saw before, and the transaction-related information is added to the WAL Buffers. Then, a separate background process will write the dirty data from this memory section into physical files known as WAL Files or Segments.

The WAL buffer area size is controlled via the **wal\_buffers** parameter. By default, the size of this area is calculated as 1/32 of the **shared\_buffers**. Changing the value requires a service restart.

### CLOG buffers

**CLOG** stands for **Commit Log**, and the CLOG Buffers are the memory space dedicated to keeping the commit status of all the concurrent transactions in the Database system. This information is essential for the **Multi-version Concurrency Control** (**MVCC**) mechanism PostgreSQL uses to handle the concurrency in the

database; we will review this in detail in another chapter. The status of a given

transaction could be any of the following:

**IN\_PROGRESS, COMMITTED, ABORTED, or SUB-COMMITTED**

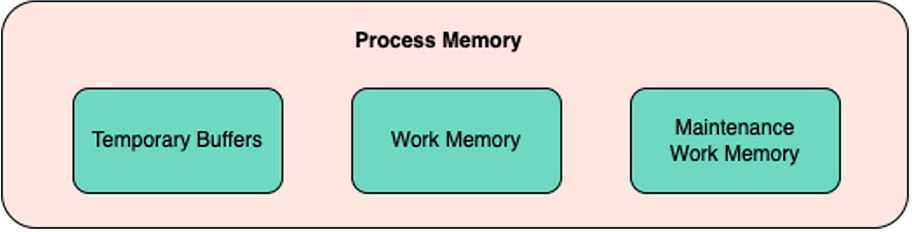
Any configuration parameter does not directly control this area’s size, but PostgreSQL

calculates it automatically.

### Process memory

The **process memory**, as its name suggests, is allocated per process. We need to consider the number of processes we expect on the system when tuning the values for these memory areas. The more processes, the more memory will be used. *Figure*

*5.2* shows the process memory sub-areas.



***Figure 5.2:*** *Process Memory areas*

#### Temporary buffers

This memory sub-area is used when one or more temporary tables are created during a user session. These are special kinds of tables visible only in the user session context; since this memory area is not shared, no other process or session can “see” the temporary tables.

The **temp\_buffers** parameter controls the size of the temporary buffer. The default size is 8MB; we can change this value dynamically, so no service restart is required.

### Work memory

The work memory sub-area is the “main” portion of memory per user session. Here all the sort and hash-table operations are processed. A sort operation can be executed when a query includes **ORDER BY** or **DISTINCT**, for example and the hash- table operations are executed with every hash-join or hash-based aggregation. If the work memory area is insufficient to process the whole operation, then PostgreSQL will use temporary files to continue the operation. Since this last involves IO calls, it is bad for performance. We will dig deeper into this in future chapters.

We can control the size of this memory sub-area with the **work\_mem** parameter. The default size is 4MB. We can modify this value dynamically.

### Maintenance work memory

This sub-area is dedicated to the maintenance tasks such as **VACUUM**, **CREATE INDEX**, or adding a **FOREIGN KEY** to a table. Multiple sessions usually do not execute these operations concurrently, so this area can be significantly larger than the work memory.

The **maintenance\_work\_mem** parameter defines the size of this area. Its default value

is 64MB, and changing it doesn’t need a service restart.

Background processes

**Client process or program (frontend):** The database frontend application performs a database action. The frontend can be a web server that wants to display a web page or a command-line tool to do maintenance tasks. PostgreSQL provides frontend tools such as psql, createdb, dropdb, and createuser.

**Server process (backend):** The server process manages database files, accepts connections from client applications, and performs actions on behalf of the client. The server process name is postgres. PostgreSQL forks a new process for each new connection; thus, client and server processes communicate with each other without the intervention of the server main process (postgres), and they have a certain lifetime that is determined by accepting and terminating a client connection.

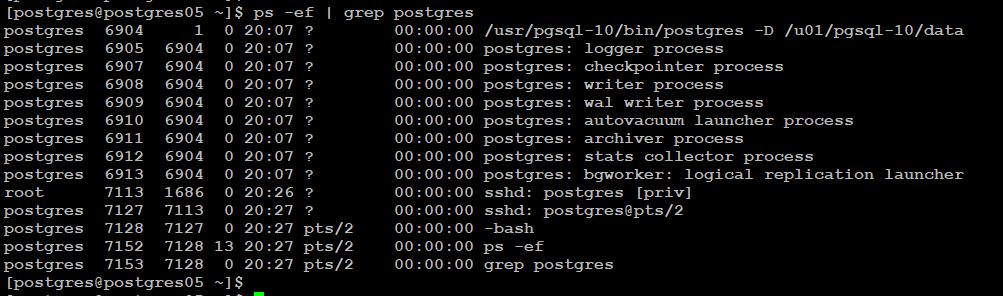
**Postmaster Process:** Postmaster process is considered as a super process in PostgreSQL.

The following are the tasks of the Postmaster process.

1. Acts as a listener to establish connections.
2. start the other background processes during the startup of the cluster.
3. Restart the background processes for any kind of termination.
4. Does instance recovery after abnormal shutdown.

**Background processes:** The following are the important background processes that are created by default.

1. logger process
2. checkpointer process
3. writer process
4. wal writer process
5. autovacuum launcher process
6. bgworker: logical replication launcher



## **PostgreSQL: logger process:**

logger process is a background process that captures log messages sent to stderr and redirects them into log files.  
The process works when logging\_collector parameter is enabled.

Set the following parameters in postgresql.conf file and restart the cluster to enable logging.

|  | log\_destination = 'stderr'  logging\_collector = on  log\_directory = 'log'  log\_filename = 'alert\_postgresql.log' |
| --- | --- |

**Example., log\_filename = 'postgresql-%d-%m-%Y.log'**

There are few more parameters for logging can be found in postgresql.conf file under the following sub sections.

**Please go thru the the file to understand more.**

* What to log
* When to log
* where to log

**postgres: checkpointer process:**

### The point at which the memory and storage to guarantee the persistence by synchronized, called a checkpoint. The pages that have been modified on a shared memory in order to create a checkpoint writes to storage.

### PostgreSQL: Background writer:

postgres: Background writer is started by the postmaster process during the start of instance.

According to the documentation, There is a separate server process called the background writer, whose function is to issue writes of “dirty” (new or modified) shared buffers. It writes shared buffers so server processes handling user queries seldom or never need to wait for a write to occur.

In other words, background writer also writes the dirty buffers to disk which helps **checkpointer** process to do less IO when CHECKPOINT occurs.(we know checkpointer process sometimes writes dirty buffers to data files.)

The main objective of background writer process is to make sure that free buffers are available for use. The algorithm that background writer process uses when writing dirty buffers is as follows.

## **PostgreSQL:WAL writer process:**

When we make changes to the data, the changes are first written to the buffers and records of these changes are written to the WAL buffer, the changes are flushed to the WAL segments when the changes are committed. Here writer process process is responsible for flushing the changes made to wal files.

To find the current wal location file, we query

postgres=# SELECT pg\_xlogfile\_name(pg\_current\_xlog\_location());

pg\_xlogfile\_name

--------------------------

00000001000000000000007C

(1 row)

postgres=#

The name is not a random collection of digits and numbers. It's comprised of three parts of 8 characters each:

00000001000000000000007C

The digits are classified as follows:

1. The first 8 digits identifies the timeline(00000001)
2. The following 8 digits identifies the (logical) xlog file(00000000)
3. The last ones represent the (physical) xlog file (Segment) (0000007C)

# postgreSQL: Autovacuum launcher

Autovacuum launcher is an optional process and it is enabled by default in PostgreSQL. This process automates the execution of vacuum and analyzes commands based on a few parameters.

If autovacuum is set, then it will wake up every **autovacuum\_naptime** seconds, and decide whether to run VACUUM, ANALYZE, or both.

But what is vacuum?

You can read in detail about vacuum in PostgreSQL

The following are the few relevant parameters that can be set in postgresql.conf file for autovacuum.(refer postgresql.conf file for more details)

**autovacuum = on**

**log\_autovacuum\_min\_duration = -1**

**autovacuum\_max\_workers = 3**

**autovacuum\_naptime = 1min**

**autovacuum\_vacuum\_threshold = 50**

**autovacuum\_analyze\_threshold = 50**

**autovacuum\_vacuum\_scale\_factor = 0.2**

**autovacuum\_analyze\_scale\_factor = 0.1**

**autovacuum\_freeze\_max\_age = 200000000**

**autovacuum\_multixact\_freeze\_max\_age =**

**autovacuum\_vacuum\_cost\_delay = 20ms**

**autovacuum\_vacuum\_cost\_limit = -1**

**postgres: bgworker: logical replication launcher**

The : logical replication launcher background process is used in logical replication purpose. If you configure publish and subscribe model, On the subscriber the “logical replication launcher” background process launched a worker process and syncs the table automatically.

**Log on a standby**

postgres 19465 19079 19 11:58 ? 00:00:04 postgres: SUBSCRIBER: bgworker: logical replication worker for subscription 16390 sync 16384

We will discuss more about the logical replication during high availability concept discussion.

## PostgreSQL Internals

## Introduction

PostgreSQL is highly efficient, and its code is very optimized. Just out of the box, it can start working and delivering a really good user experience, like a *magic* box that receives queries and returns data. This is possible because multiple subsystems work cooperatively and coordinately, each responsible for a specific part of the process. Knowing what is inside PostgreSQL and how it works is essential in the DBA’s path.

## Structure

In this chapter, we will learn about the internals of PostgreSQL. This chapter is going to cover following main concepts:

* ACID
* MVCC
* Vacuum
* Transaction Isolation Levels
* Query Processing

## Objectives

In this chapter, we will learn the different internal subsystems and concepts involved in the PostgreSQL functionality. Once you complete it, you will relate the different stages from a user session starting a query until the result is gotten. How they affect each other in a concurrent system, and the different subsystems that enable and maintain PostgreSQL ready to operate with multiple user sessions at the best performance.

## ACID

The full form of ACID is **Atomicity, Consistency, Isolation, and Durability**. All RDBMS majorly follow ACID properties to manage the transaction level. Let us try to understand each property.

### Atomicity

PostgreSQL makes sure that transactions are fully complete, or they fail entirely in case of any failure. It does not leave the transaction state in between or partially complete state.

### Consistency

At any point, transactions are in a consistent state and adhere to a set of rules defined. At any given time, data integrity checks (constraints) maintain consistency within the database.

### Isolation

Concurrent transactions happen within the database. Isolation property makes sure that each transaction is separate and maintains its state within that transaction. One transaction cannot interfere with any other transaction unless any input or output, or processing needs to be shared amongst the transactions.

### Durability

At any given point, database transactions, once committed or rollbacked, must maintain as it is, no matter whether the same DB is accessed at the same time or years after the transaction. It should be durable and show the same result despite failovers occurring in the Data Center or Database. If there are any corruptions or

any other unavoidable circumstance then we may not be able to access the data but otherwise data should be constant irrespective of when or from where it has been accessed.

## MVCC

The Full form of MVCC is **MultiVersion Concurrency Control**. MVCC Architecture of PostgreSQL ensures that all ACID properties are compliant. At any given time, whenever concurrent transactions are performed on the same DB Objects or different DB Objects, DB will be in the same consistent state, and all transactions are isolated.

Readers do not block the writers, and writers do not block readers either. While querying any DB Object, PostgreSQL ensures that DB is in the same consistent state as it was a while ago unless any change has occurred. Multiple copies of DB Objects are created in case the same table/view is being accessed concurrently. The moment transaction gets committed, then the object gets updated, and from the next time onward, the updated records are visible to the user.

Each transaction is given transaction id (*xid*), and if the current *xid* > the *xid* which was committed a while ago, then the current *xid* can see the updated records.

Let us try to understand the same using an example. Let us say that Account A has 2 Debit Cards. One debit card is used by the mother and another by the child studying at a different location. As mentioned earlier, the underlying account is the same for both debit cards. It can very well happen that both mother and child can access the same account using both their own debit cards at the same time concurrently.

Both will be reading the same account balance as readers do not block other readers. When both will try to withdraw money from ATMs, a writer will block the other writer. In this scenario, either both will be declined to withdraw money as that might create inconsistent data, or one transaction will be successful, and the other will not be as both are reading the same data while inserting ATM Debit Card, but when both will withdraw the money that time the transaction value should be withdrawn from both cards and balance needs to be updated accordingly.

Say the Balance is $1000, and withdrawing $100 from each card should make the balance $1000-$100-$100 = $800. If ACID & MVCC are not followed, both will be shown $1000-$100 =$900, which is incorrect. That is why due to MVCC, either both will be declined, or only one person will be successful. Then the ones whose transaction declined will try a second time; he/she will be shown the balance as

$900, and then withdrawing $100 will make the final balance $800.

In this way, transactions are in a consistent, durable, and isolated state at any point so that MVCC get accomplished.

We will see transaction management in more detail further in this chapter.

## Vacuum

This is one of the routine maintenance activities for PostgreSQL Databases. To understand the vacuum, let us first try to understand how the **UPDATE** and **DELETE** of a row work in PostgreSQL databases.

At the time of an **UPDATE** operation, the PostgreSQL database performs two major

steps:

1. It inserts new records with updated values.
2. It marks the existing row for deletion and deactivates the same.

Eventually, when a user selects the records, only updated records are visible. At the time of **DELETE**, rows are marked for deletion and deactivated but physically not removed.

In this way, all the rows marked for deletion are known as *dead tuples*. These *dead tuples* in the PostgreSQL database, are removed physically by performing **VACUUM** activity.

**VACUUM** can be compared with the fragmentation and defragmentation concept of the Operating System where internal disk space is formatted by cleaning each block of disk space. This process helps in freeing the space within the data files pages.

The vacuum operations can be executed on the database from different sources and

levels. In the following subsections, we will learn about them.

### Autovacuum

As the name suggests, it is automatically performed by PostgreSQL, and by default, autovacuum is enabled in **postgresql.conf** file. However, this is an optional setting the PostgreSQL community highly recommends this setting be kept on.

Autovacuum is controlled by the autovacuum process of PostgreSQL, which is active when the autovacuum is on. Autovacuum is performed as per the configuration set in the configuration file. One can customize the same for specific tables and indexes. Autovacuum will help remove *dead tuples*, and it utilizes that space released by the removal of *dead tuples* whenever new DML operations are performed on the database.

It will not free the disk space back to the operating system; however, DB/Object Size will have that space to be used for new data. This is purely an online activity that helps in increasing performance while querying the tables.

### VACUUM FULL

This is a manual process that needs downtime for that particular object on which the VACUUM FULL is running. It can be executed on the Database Level or Table Level. This helps in giving space back to Operating System. Bloated data or dead tuples are completely removed to free the space.

In the background, performing VACUUM FULL creates a copy of the DB object and copies only the active or required data. Toward the end, it will remove the old table with the bloated size and keep the new one without dead tuples. That is why the object on which the VACUUM FULL is being performed becomes inaccessible till the vacuum is completed.

Also, at the start of the activity, a pre-requisite VACUUM FULL needs to double the table size. For example, if any table is of 50 GB size, then a minimum of 100 GB disk space is needed to perform VACUUM FULL on that table. At the end of the activity, the table size will reduce depending upon the bloat data size of the table. That is why one must check the bloats in the table before performing VACUUM FULL. If bloats are less or negligible, then it is not recommended to perform VACUUM FULL on the table.

### Manual VACUUM

It is the same as autovacuum, with the only difference that it will not be triggered

automatically and needs to be performed manually.

Please find the most frequent options used with **VACUUM**:

* **FREEZE**: The age of the table can be reset using this option.
* **VERBOSE**: It prints the detailed log of each stage, metadata count, and so on. details on the screen.
* **ANALYZE**: It will update the metadata which can be helpful for the query

planner while executing/processing any query.

Whenever the vacuum is running, the below queries might help to get details of the ongoing maintenance of the DB.

SELECT now()-query\_start age, \* FROM pg\_stat\_activity where query ILIKE ‘%vacuum%’;

This query gives the current vacuum without disclosing at which phase of the vacuum is currently in progress. Rather it can give the details like when was vacuum started, from which IP/Host and the like.

SELECT relid::regclass,\* FROM pg\_stat\_progress\_vacuum;

This query gives details about the different phases of the vacuum:

* Initializing
* Scanning heap
* Vacuuming indexes
* Vacuuming heap
* Cleaning up indexes
* Truncating heap

Performing final cleanup(*Reference: Progress Reporting*)

SELECT relname,last\_vacuum, last\_autovacuum, last\_analyze, last\_ autoanalyze

FROM pg\_stat\_user\_tables

WHERE relname = ‘<table\_name>’;

This query gives details of the last vacuum performed on a given table.

#### pg\_repack

This is an open-source extension that can be used instead of VACUUM FULL without minimal downtime. This helps reclaim the disk space and occupy the lock for a very short duration (usually in milliseconds or seconds) at the start and end of the activity.

When the **pg\_repack** command is issued on any table or index, it will acquire a lock to create an initial copy of that object. It will remove the dead tuples on the object copy, and at the end of the activity, it will again acquire a lock that will rename the new table to the existing name and process the data which was modified during the repack process.

This is one of the good options as an alternative to VACUUM FULL. When **pg\_ repack** is in progress, DDL operations cannot be performed on the table as it holds an access share lock during the repack process.

### Preventing transaction ID wraparound failures

MVCC transaction semantics depend on comparing transaction ID (*xid*) numbers: a row version with an insertion *xid* greater than the current transaction’s *xid* is “in the future” and should not be visible to the current transaction. But since transaction IDs have a limited size (32 bits), a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound.

To avoid this, every table in the database must be vacuumed at least once every billion transactions. Autovacuum, if enabled, takes care of transaction wraparound issues.

## Transaction isolation levels

As we learned in the first section of this chapter, the “I” from the ACID acronym stands for Isolation. This is one of the features the SQL standard defines for a **Relational Database Management System (RDBMS**) and means how the concurrent transaction affects each other.

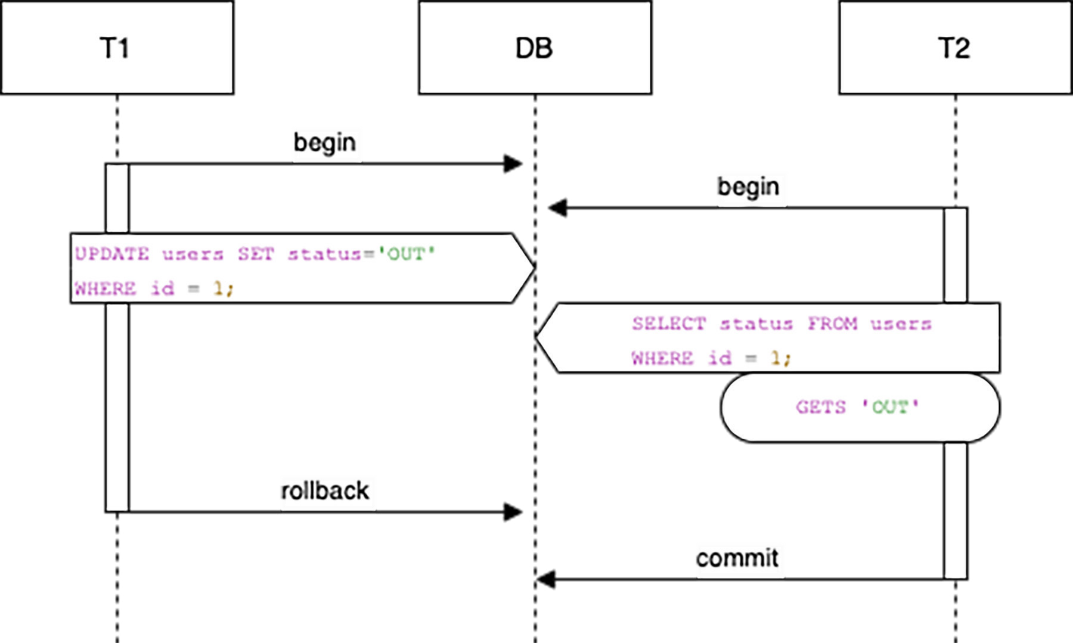
PostgreSQL allows four transaction isolation levels: read uncommitted, read committed, repeatable read, and serializable. Each guarantees transactions free from the different effects of other concurrent transactions, known as phenomena. Let us start reviewing the phenomena and their effects to understand these concepts better.

### Phenomena

There are also four possible phenomena types: dirty read, nonrepeatable read, phantom read, and serialization anomaly.

### Dirty read

The dirty read refers to the effect of one transaction being able to read the uncommitted changes from other transactions. In PostgreSQL, this is not possible at any isolation level. *Figure 6.1* illustrates how this would affect two concurrent transactions:



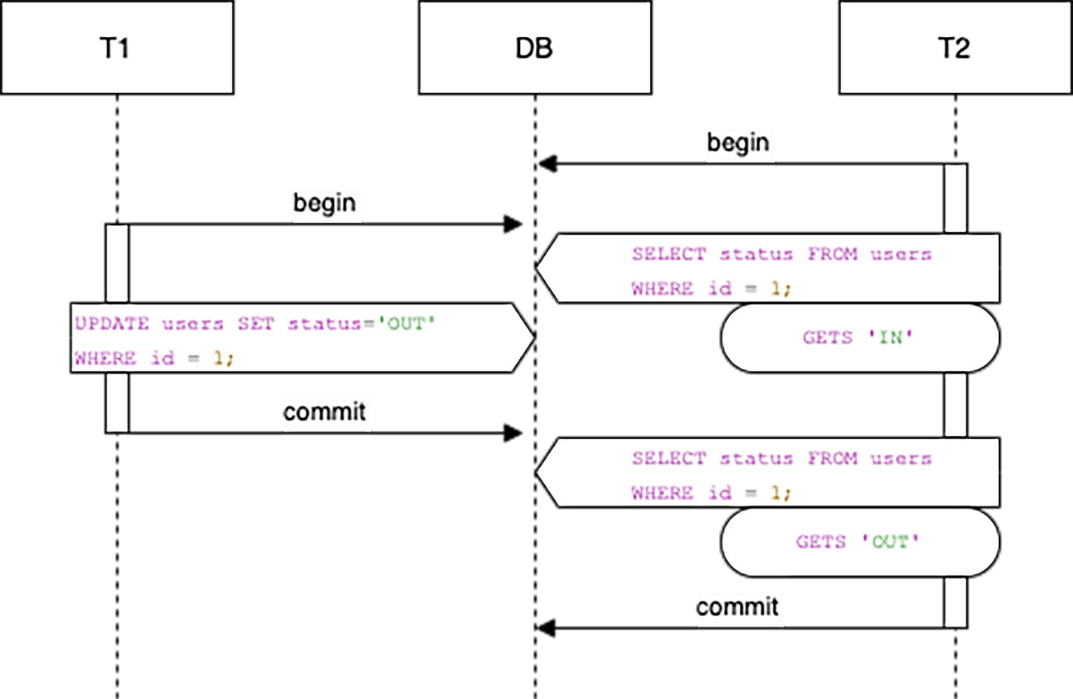
***Figure 6.1:*** *Dirty reads*

### Non-repetable read

This refers to the effect of one transaction reading different results from the same

query during its duration. This is because other transactions commit changes. *Figure*

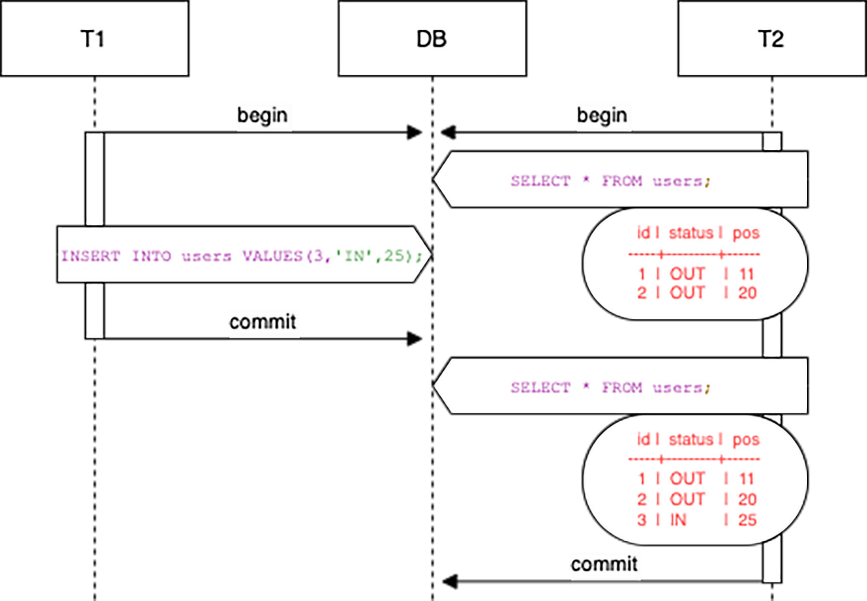
*6.2* shows how this looks:



***Figure 6.2:*** *Nonrepetable reads*

### Phantom read

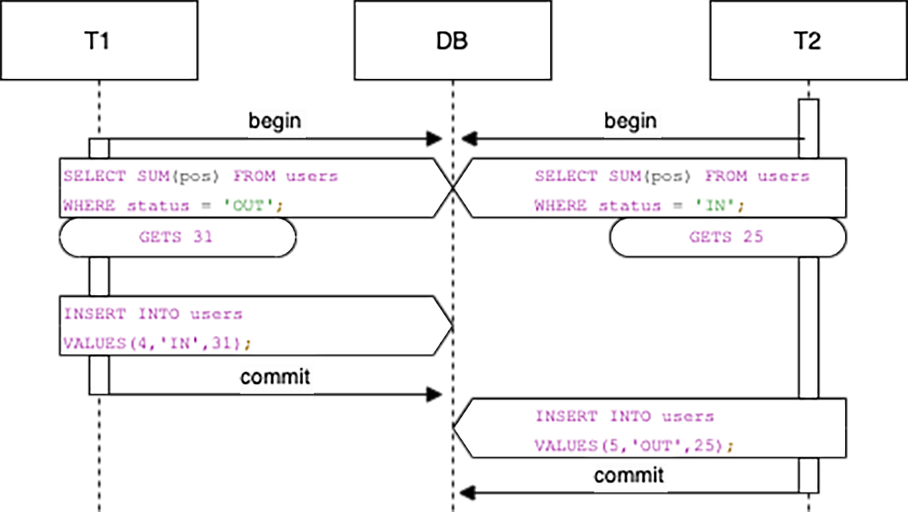
This means one transaction reading a different set of rows from one previous read. This is because another transaction has added or removed rows from the resulting set and has committed these changes. *Figure 6.3* illustrates the flow of this:



***Figure 6.3:*** *Phantom read*

### Serialization anomaly

This refers to the effect of successfully committing a group of transactions whose final result can not be reproduced by executing the transactions one at a time in any order:



***Figure 6.4:*** *Serialization anomaly*

As shown in *Figure 6.4*, each transaction executes a **SELECT** to get the summary of the column **pos**; the first filters by the column **status** equal to ‘OUT’, and the second filters by **status** equal to ‘IN’. Then each one inserts a new row setting the column **pos** value the same as the result of their initial **SELECT** and with the opposite value for the **status** column. It is not possible to get the same results for each statement executing the transactions one by one since the initial result for the **SELECT** will be different depending on which **INSERT** happens first.

The following are the four sub folders created under postgres home

| include | Sets the directory for installing C and C++ header files. The default is ***PREFIX***/include. |
| --- | --- |
| share | Sets the root directory for various types of read-only data files.The default is ***PREFIX***/share. |
| lib | PostgreSQL Library directory refers to the location where the PostgreSQL library files reside. Postgres Lib directory path must contain libpq.so file and other related files. This library is used to connect to the database and execute queries. |
| bin | Specifies the directory for executable programs. The default is ***EXEC-PREFIX***/bin, which normally means /usr/local/pgsql/bin |

Here, bin is the important directiry where the executables are stored. We will try to understand the contents of bin directory in detail.

| executable | Purpose |
| --- | --- |
| postmaster -> postgres | postmaster process |
| postgres | postmaster process |
| pg\_recvlogical | pg\_recvlogical controls logical decoding replication slots and streams data from such replication slots. |
| pg\_receivewal | pg\_receivewal is used to stream the write-ahead log from a running PostgreSQL cluster. The write-ahead log is streamed using the streaming replication protocol, and is written to a local directory of files. This directory can be used as the archive location for doing a restore using point-in-time recovery |
| pg\_ctl | controls the postgres cluster |
| pg\_controldata | get the pg\_control data |
| pg\_config | pg\_config — retrieve information about the installed version of PostgreSQL |
| pg\_basebackup | take physical backup |
| pg\_archivecleanup | Clear archivelogs |
| initdb | initialize the database cluster |
| ecpg | ecpg is the embedded SQL preprocessor for C programs. It converts C programs with embedded SQL statements to normal C code by replacing the SQL invocations with special function calls. The output files can then be processed with any C compiler tool chain. |
| vacuumdb | vacuum the database |
| reindexdb | reindexdb is a utility for rebuilding indexes in a PostgreSQL database. |
| psql | connect to postgres database |
| pg\_waldump | view the wal file contents |
| pg\_upgrade | upgrade your cluster |
| pg\_test\_timing | pg\_test\_timing is a tool to measure the timing overhead on your system and confirm that the system time never moves backwards. |
| pg\_test\_fsync | pg\_test\_fsync is intended to give you a reasonable idea of what the fastest wal\_sync\_method is on your specific system, as well as supplying diagnostic information in the event of an identified I/O problem. |
| pg\_rewind | rewind the cluster after switch over |
| pg\_restore | import logical backup |
| pg\_resetwal | reset the wal file |
| pg\_isready | pg\_isready is a utility for checking the connection status of a PostgreSQL database server. The exit status specifies the result of the connection check. |
| pg\_dumpall | used for logical backup or exporting of entire cluster |
| pg\_dump | used for logical backup or exporting |
| pgbench | benchmark testing |
| dropuser | drop the user |
| dropdb | drop the database |
| createuser | create the user |
| createdb | create the database |
| clusterdb | clusterdb is a utility for reclustering tables in a PostgreSQL database. It finds tables that have previously been clustered, and clusters them again on the same index that was last used. Tables that have never been clustered are not affected. |
| vacuumlo | vacuumlo is a simple utility program that will remove any “orphaned” large objects from a PostgreSQL database. An orphaned large object (LO) is considered to be any LO whose OID does not appear in any oid or lo data column of the database. |
| pg\_standby | pg\_standby supports creation of a “warm standby” database server. It is designed to be a production-ready program, as well as a customizable template should you require specific modifications. |
| oid2name | oid2name is a utility program that helps administrators to examine the file structure used by PostgreSQL. |

Data directory layout

| **Item** | **Description** |
| --- | --- |
| PG\_VERSION | A file containing the major version number of PostgreSQL |
| base | Subdirectory containing per-database subdirectories |
| current\_logfiles | File recording the log file(s) currently written to by the logging collector |
| global | Subdirectory containing cluster-wide tables, such as pg\_database |
| pg\_commit\_ts | Subdirectory containing transaction commit timestamp data |
| pg\_dynshmem | Subdirectory containing files used by the dynamic shared memory subsystem |
| pg\_logical | Subdirectory containing status data for logical decoding |
| pg\_multixact | Subdirectory containing multitransaction status data (used for shared row locks) |
| pg\_notify | Subdirectory containing LISTEN/NOTIFY status data |
| pg\_replslot | Subdirectory containing replication slot data |
| pg\_serial | Subdirectory containing information about committed serializable transactions |
| pg\_snapshots | Subdirectory containing exported snapshots |
| pg\_stat | Subdirectory containing permanent files for the statistics subsystem |
| pg\_stat\_tmp | Subdirectory containing temporary files for the statistics subsystem |
| pg\_subtrans | Subdirectory containing subtransaction status data |
| pg\_tblspc | Subdirectory containing symbolic links to tablespaces |
| pg\_twophase | Subdirectory containing state files for prepared transactions |
| pg\_wal | Subdirectory containing WAL (Write Ahead Log) files |
| pg\_xact | Subdirectory containing transaction commit status data |
| postgresql.auto.conf | A file used for storing configuration parameters that are set by ALTER SYSTEM |
| postmaster.opts | A file recording the command-line options the server was last started with |
| postmaster.pid | A lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (empty on Windows), first valid listen\_address (IP address or \*, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown) |

Host Based Autentication file (pg\_hba.conf)

Client authentication is controlled by a configuration file, which traditionally is named pg\_hba.conf and is stored in the database cluster's data directory. (HBA stands for host-based authentication.) A default pg\_hba.conf file is installed when the data directory is initialized by initdb. It is possible to place the authentication configuration file elsewhere, however; see the hba\_file configuration parameter.

How do you enable access to external connections with md5 authentication?

host all all 0.0.0.0/0 md5

How do you restrict DEMOUSER database to reject connections?

local all DEMOUSER 0.0.0.0/0 reject

host all all 0.0.0.0/0 md5

**[postgres@postgres01 10]$ psql -U demouser -d postgres**

**psql: FATAL: pg\_hba.conf rejects connection for host "[local]", user "demouser", database "postgres"**

**[postgres@postgres01 10]$**

How do you restrict DEMOUSER database to reject connections if it is from 192.168.1.4?

host all DEMOUSER 192.168.1.4 reject

host all all 0.0.0.0/0 md5

How do you ask DEMOUSER to authenticate and rest ignore authentication?

host all DEMOUSER 0.0.0.0/0 md5

host all all 0.0.0.0/0 trust

How do you set up hba configuration for replica user to take backup from remote machine?

host replication replica 0.0.0.0/0 md5

**Databases, Users and Schemas**

**PostgreSQL Databases**

Every instance of a running PostgreSQLserver manages one or more databases. Databases are therefore the topmost hierarchical level for organizing SQLobjects (**“**database objects”). 

## Creating a Database

CREATE DATABASE ***name***;

## Template Databases

CREATE DATABASEactually works by copying an existing database. By default, it copies the standard system database named template1.Thus that database is the **“**template”from which new databases are made. If you add objects to template1, these objects will be copied into subsequently created user databases. 

There is a second standard system database named template0.This database contains the same data as the initial contents of template1, that is, only the standard objects predefined by your version of PostgreSQL. template0should never be changed after the database cluster has been initialized

## Destroying a Database

Databases are destroyed with the command DROP DATABASE.

**drop database name;**

*Data dictionary view* ***pg\_database***

**Viewing the List of Databases**

SELECT datname from pg\_database;

**Creating and Managing Schemas**

Schemas logically organize objects and data in a database. Schemas allow you to have more than one object (such as tables) with the same name in the database without conflict if the objects are in different schemas.

**The Default “Public” Schema**

Every database has a default schema named *public*. If you do not create any schemas, objects are created in the *public* schema. All database roles (users) have CREATE and USAGE privileges in the *public* schema. When you create a schema, you grant privileges to your users to allow access to the schema.

**Creating a Schema**

Use the CREATE SCHEMA command to create a new schema. For example:

=> CREATE SCHEMA myschema;

**Dropping a Schema**

Use the DROP SCHEMAcommand to drop (delete) a schema. For example:

=> DROP SCHEMA myschema;

By default, the schema must be empty before you can drop it. To drop a schema and all of its objects (tables, data, functions, and so on) use:

=> DROP SCHEMA myschema CASCADE;

**Schema Search Paths**

To specify an object’s location in a database, use the schema-qualified name. For example:

=> SELECT \* FROM myschema.mytable;

You can set the search\_path configuration parameter to specify the order in which to search the available schemas for objects. The schema listed first in the search path becomes the *default* schema. If a schema is not specified, objects are created in the default schema.

**Setting the Schema Search Path**

The search\_path configuration parameter sets the schema search order. The ALTER DATABASE command sets the search path. For example:

=> ALTER DATABASE mydatabase SET search\_path TO myschema, public, pg\_catalog;

You can also set search\_path for a particular role (user) using the ALTER ROLE command. For example:

=> ALTER ROLE sally SET search\_path TO myschema, public, pg\_catalog;

**Viewing the Current Schema**

Use the current\_schema() function to view the current schema. For example:

=> SELECT current\_schema();

Use the SHOW command to view the current search path. For example:

=> SHOW search\_path;

PostgreSQL default schemas

1. information\_schema
2. pg\_catalog
3. pg\_toast
4. public

information\_schema consists of a standardized set of views that contain information about the objects in the database. These views get system information from the system catalog tables in a standardized way.

pg\_catalog contains the system catalog tables, built-in data types, functions, and operators. It is always part of the schema search path, even if it is not explicitly named in the search path.

pg\_toast stores large objects such as records that exceed the page size. This schema is used internally by the Greenplum Database system.

Every database has a default schema named *public*. If you do not create any schemas, objects are created in the *public* schema.

## Database Roles

PostgreSQL manages database access permissions using the concept of *roles*. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables and functions) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant *membership* in a role to another role, thus allowing the member role to use privileges assigned to another role.

To create a role use the CREATE ROLE SQL command:

CREATE ROLE name;

To remove an existing role, use the analogous DROP ROLE command:

DROP ROLE name;

For convenience, the programs createuser and dropuser are provided as wrappers around these SQL commands that can be called from the shell command line:

createuser name

dropuser name

To determine the set of existing roles, examine the pg\_roles system catalog, for example

SELECT rolname FROM pg\_roles;

## Role Attributes

## database role can have a number of attributes that define its privileges and interact with the client authentication system.

login privilege

superuser status

database creation

role creation

initiating replication

password

## Role Membership

revoked from, a group as a whole. In PostgreSQL this is done by creating a role that represents the group, and then granting membership in the group role to individual user roles.

To set up a group role, first create the role:

CREATE ROLE name;

Typically a role being used as a group would not have the LOGIN attribute, though you can set it if you wish.

Once the group role exists, you can add and remove members using the GRANT and REVOKE command:

GRANT group\_role TO role1, ... ;

REVOKE group\_role FROM role1, ... ;

Following are the commands to check the role details.

**\du**

**pg\_roles**

[ACTIVITY]

Create two roles, dba and developer and create two developers and two dba’s and now assign read, write permission on production tables to dba role and read only access to developer role.

Also, track the user activity in log.

*[I already have a database “airportdb” with owner “airport” and has a table “emp” in airport schema]*

Create role airport\_read;

Create role airport\_ps;

GRANT CONNECT ON DATABASE airportdb TO airport\_read;

GRANT USAGE ON SCHEMA airport TO airport\_read;

GRANT SELECT ON ALL TABLES IN SCHEMA airport TO airport\_read;

ALTER DEFAULT PRIVILEGES IN SCHEMA airport GRANT SELECT ON TABLES TO airport\_read;

GRANT CONNECT ON DATABASE airportdb TO airport\_ps;

GRANT USAGE ON SCHEMA airport TO airport\_ps;

GRANT ALL ON ALL TABLES IN SCHEMA airport TO airport\_ps;

ALTER DEFAULT PRIVILEGES IN SCHEMA airport GRANT ALL ON TABLES TO airport\_ps;

GRANT ALL ON ALL SEQUENCES IN SCHEMA airport TO airport\_ps;

ALTER DEFAULT PRIVILEGES IN SCHEMA airport GRANT ALL ON SEQUENCES TO airport\_ps;

GRANT ALL ON ALL FUNCTIONS IN SCHEMA airport TO airport\_ps;

ALTER DEFAULT PRIVILEGES IN SCHEMA airport GRANT ALL ON FUNCTIONS TO airport\_ps;

create user ab\_dba1 with password 'ab\_dba1';

create user ab\_dba2 with password 'ab\_dba2';

create user ab\_dev1 with password 'ab\_dev1';

create user ab\_dev2 with password 'ab\_dev2';

grant airport\_ps to ab\_dba1;

grant airport\_ps to ab\_dba2;

grant airport\_read to ab\_dev1;

grant airport\_read to ab\_dev2;

alter user ab\_dba1 set search\_path=airport, public,ab\_dba1;

alter user ab\_dba2 set search\_path=airport, public,ab\_dba2;

alter user ab\_dev1 set search\_path=airport, public,ab\_dev1;

alter user ab\_dev2 set search\_path=airport, public,ab\_dev2;

alter user ab\_dba1 set log\_statement='all';

alter user ab\_dba2 set log\_statement='all';

alter user ab\_dev1 set log\_statement='all';

alter user ab\_dev2 set log\_statement='all';

[LOGIN AND CHECK IF EVERYTHING IS FINE]

**Tablespaces**

<https://pgdash.io/blog/tablespaces-postgres.html>

<https://www.cybertec-postgresql.com/en/when-to-use-tablespaces-in-postgresql/>

**Enabling archivelog mode**

Edit the following parameters in postgresql.conf file and restart the cluster.

**archive\_mode = on**

**archive\_command = 'cp %p /u01/archives/%f**

**archive\_timeout = 5min**

PostgreSQL Backup Methods

PostgreSQL offers two types of backup methods:

* Logical backups
* Physical backups

**Logical backups** are like snapshots of a database. These are created using the pg\_dump or pg\_dumpall utility that ships with PostgreSQL. Logical backups:

* Back up individual databases or all databases
* Back up just the schemas, just the data, individual tables, or the whole database (schemas and data)
* Create the backup file in proprietary binary format or in plain SQL script
* Can be restored using the pg\_restore utility which also ships with PostgreSQL
* Do not offer point-in-time recovery (PITR)

**Physical backups** are different from logical backups because they deal with binary format only and makes file-level backups. Physical backups:

* Offer point-in-time recovery
* Back up the contents of the PostgreSQL *data directory* and the *WAL* (Write Ahead Log) files
* Use the PostgreSQL pg\_start\_backup and pg\_stop\_backup commands. However, these commands need to be scripted, which makes physical backups a more complex process (pg\_basebackup).

**Physical Backups**

**pg\_base\_backup**

The easiest way to perform a base backup is to use the pg\_basebackup tool. It can create a base backup either as regular files or as a tar archive. If more flexibility than pg\_basebackup can provide is required, you can also make a base backup using the low level API which discussed earlier.

Command

***pg\_basebackup -D <location\_of\_backup\_file> -Ft -z -U postgres -w***

[postgres@postgres02 backups]$**pg\_basebackup -D /u01/pgsql/backups -F t -z -U postgres -w -v**

pg\_basebackup: initiating base backup, waiting for checkpoint to complete

pg\_basebackup: checkpoint completed

pg\_basebackup: write-ahead log start point**: 1/C1000028** on timeline 2

pg\_basebackup: starting background WAL receiver

pg\_basebackup: write-ahead log end point: 1/C1000130

pg\_basebackup: waiting for background process to finish streaming ...

pg\_basebackup: base backup completed

[postgres@postgres02 backups]$

To make use of the backup, you will need to keep all the WAL segment files generated during and after the file system backup. To aid you in doing this, the base backup process creates a *backup history file* that is immediately stored into the WAL archive area. This file is named after the first WAL segment file that you need for the file system backup.

**Setting up a new redo log location**

Step 1: stop database.

Step 2: create pg\_xlog directory at different location and copy pg\_xlog files from data directory.

Step 3: delete pg\_xlog directory and create a symbolic link at data directory.

*ln -s /u02/New\_PostgreData/pg\_xlog/ /u02/New\_PostgreData\_1/*

## Recovery Target Settings

**By default, recovery will recover to the end of the WAL log.** The following parameters can be used to specify an earlier stopping point. At most oneof recovery\_target, recovery\_target\_lsn, recovery\_target\_name, recovery\_target\_time, or recovery\_target\_xid can be used; if more than one of these is specified in the configuration file, the last entry will be used.

recovery\_target = 'immediate' -- consistent state has been reached

recovery\_target\_lsn= '0/3019838' -- select pg\_current\_wal\_lsn();

recovery\_target\_time=2019-10-26 19:34:55

recovery\_target\_xid = ‘634’ -- select txid\_current();

Logical Backups:

# pg\_dump

pg\_dump is a utility for backing up a PostgreSQL database. It makes consistent backups even if the database is being used concurrently. pg\_dump does not block other users accessing the database (readers or writers).

Dumps can be output in script or archive file formats. Script dumps are plain-text files containing the SQL commands required to reconstruct the database to the state it was in at the time it was saved. To restore from such a script, feed it to psql. Script files can be used to reconstruct the database even on other machines and other architectures; with some modifications, even on other SQL database products.

The alternative archive file formats must be used with pg\_restore to rebuild the database. They allow pg\_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive file formats are designed to be portable across architectures.

pg\_dump - -help

or

man pg\_dump

# pg\_restore

pg\_restore is a utility for restoring a PostgreSQL database from an archive created by pg\_dump in one of the non-plain-text formats. It will issue the commands necessary to reconstruct the database to the state it was in at the time it was saved. The archive files also allow pg\_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are designed to be portable across architectures.

pg\_restore can operate in two modes. If a database name is specified, pg\_restore connects to that database and restores archive contents directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created and written to a file or standard output. This script output is equivalent to the plain text output format of pg\_dump. Some of the options controlling the output are therefore analogous to pg\_dump options.

Obviously, pg\_restore cannot restore information that is not present in the archive file. For instance, if the archive was made using the "dump data as INSERT commands" option, pg\_restore will not be able to load the data using COPY statements.

*Example.,* pg\_restore -d newdb db.dump

Examples.,

Take flights table backup with pg\_dump in plain text format.

**pg\_dump -U demouser -d demo -t flights> demo\_plain.sql**

Take flights table backup with pg\_dump in plain text format with –Fp option.

**pg\_dump -U demouser -d demo -Fp -t flights -f demo\_plain.bkp**

Take flights table backup with pg\_dump in tar format with –Ft option.

**pg\_dump -U demouser -d demo -Ft -t flights -f demo\_plain.tar**

Take flights table backup with pg\_dump in custorm format with –Fc option.

**pg\_dump -U demouser -d demo -Fc -t flights -f demo\_plain.custom**

Take flights, seats table backup with pg\_dump in directory format with –Fd option.

**pg\_dump -U demouser -d demo -Fd -t flights -t sets -f twotables**

Take flights, seats table backup with pg\_dump in directory format with 4 parallel processes.

**pg\_dump -U demouser -d demo -Fd -t flights -t sets -f parallel4 -j 4**

Take flights table backup with pg\_dump in custorm format with 4 compression.

**pg\_dump -U demouser -d demo -Fc -t flights -f demo\_plain.custom -Z 3**

Restore commands

Restore table flights from demo\_plain.custom

**pg\_restore -U demouser -d demo demo\_plain.custom**

**check :**

psql -U demouser -d demo -c "select count(1) from flights";

Create a dababase demoview and import all functions from backup fullmeta.bkp

**psql -c "create database demoview with owner demouser";**

**psql -U demouser -d demoview -c "create schema demouser";**

**pg\_restore -l fullmeta.bkp | grep FUNCTION > function\_list**

**pg\_restore -U demouser -d demoview -L function\_list fullmeta.bkp**

Restore a single table from backup

**pg\_restore -U demouser -d demoview -t flights fulldata.bkp**

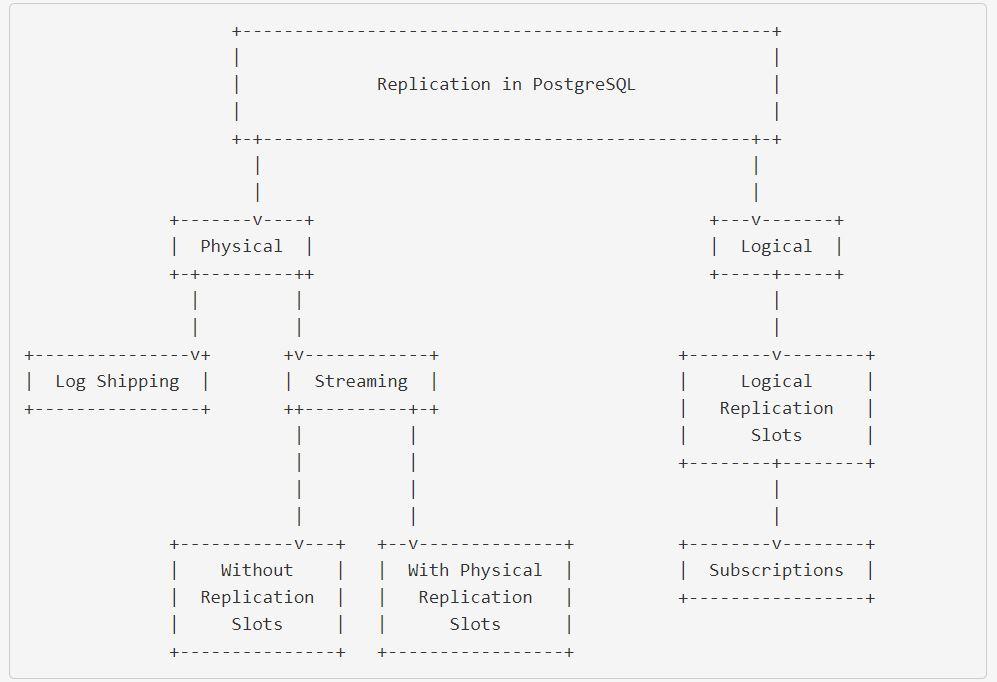
**PostgreSQL Replication**

PostgreSQL comes with various ways of replicating data between servers, along with configuration options to tweak and tune the replication process to fit your needs. Regardless of what type of replication you set up, it is important to monitor all endpoints of replication to ensure that your data is safe and sound.

Read on to learn more about monitoring replication in Postgres.

Types of Replication

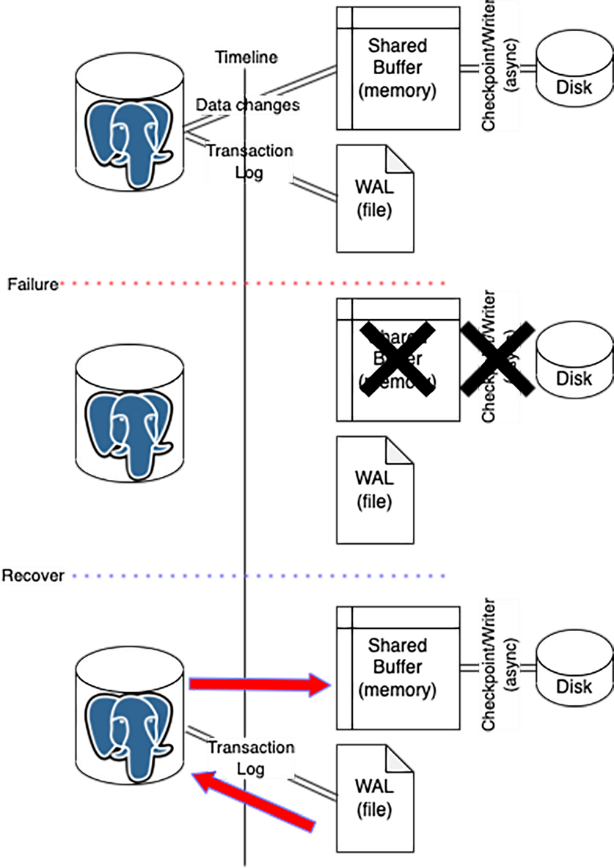
Broadly, there are 2 types of replication in PostgreSQL – **physical** and **logical**. Here is a quick overview picture:



## Physical replication

We could say that physical replication is the “original” replication method supported by PostgreSQL. It became possible because of a couple of mechanisms we already have reviewed: the **Write-Ahead Log** (**WAL**) and the continuous archiving; you can look at ***Chapter 5, Architecture of PostgreSQL*** if you want to refresh the concepts.

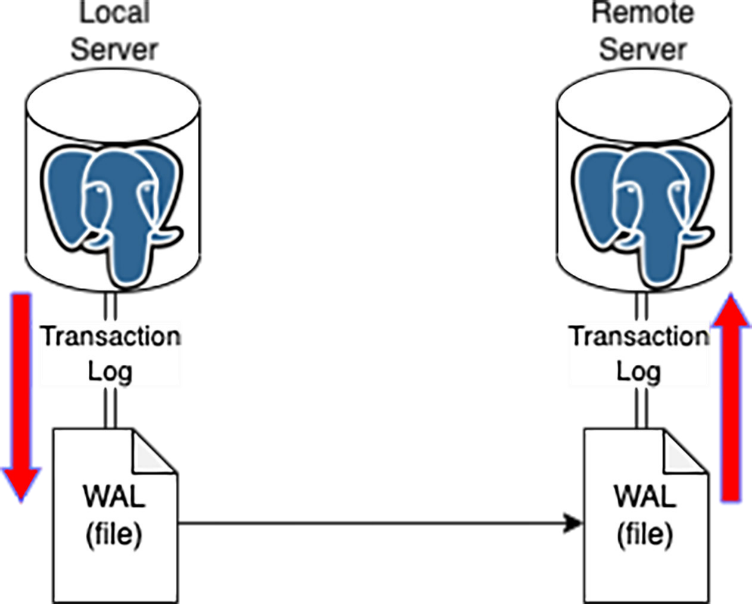
The WAL and the continuous archiving were designed and added to postgres to enable the possibility of performing *recovery*. If you remember, PostgreSQL works in memory, all the data changes happen in the shared buffers area, and they are not flushed to disk immediately but asynchronously. So, in the event of a power or hardware failure, these memory changes might get lost. However, the WAL keeps a record of all the transactions that have changed data, which are flushed to the disk on every commit. In the same failure event where all the memory data is lost, PostgreSQL can perform *recovery* and *redo* all the data changes from the records in the WAL. *Figure 8.1* illustrates the recovery process.



***Figure 8.1:*** *PostgreSQL recovery after a failure.*

Continuous archiving also brings extra protection since the WAL files may get removed or reused over time, and archiving them to an external or dedicated device covers this situation, then postgres will still be able to recover.

These functionalities are great for getting our system up and running after the failure and, more importantly, with no data loss. But this same logic can be applied in a different server remotely, right? The answer is yes! The same way the data changes are replied to after a crash to *redo* the database state can be done continuously in a remote server on top of a restored physical backup. *Figure 8.2* shows in a very basic way how this looks.



***Figure 8.2:*** *PostgreSQL basic physical replication diagram.*

So, the physical replication can be done by re-applying the WAL changes from a read/write (primary) instance on top of a physical data copy (backup) in a remote (standby) instance.

As you can think, there are a few requirements to getting this working properly, and there are a couple of *variations* of the physical replication and some features to build different topologies. We will learn about all these details next.

**Physical replication requirements:**

As we saw in the previous *Chapter 7, Backup and Restore in PostgreSQL*, the physical backups are an exact binary copy of a running PostgreSQL, so they are inconsistent, and we need the WAL to make them consistent during a restore. And, as we stated before, the physical replication is an “extension” of this logic, so we can say the next

are the minimum requirements to build a replica server or standby in postgres argot.

Consider the read/write instance or primary already should exist.

* A server with enough resources (CPU and memory). This depends on the

replica’s usage and whether this is supposed to handle any read workload.

* The server requires enough storage capacity according to the DB size.
* The same filesystem/directory layout as the primary is expected.
* A PostgreSQL server running in the same version as the primary.
* Even when it is not a “must,” using the same OS for the replica as the primary

node, including the same kernel version, can prevent unsightly side effects.

* The replica node should be able to directly access the WAL archive repository or the primary server.
* If the WAL archive repository is the same as the backups, then the replica can be built from a backup, and there would not be a need to reach the primary server.

### Hot standby

During a physical backup restoration, the data cannot be read. Postgres needs to apply all the required WAL and then “open” the database at a consistent point so it can be acceded for reads and writes.

Since physical replication follows essentially the same principles, just keeping the restore operation going, the same effect of the data not being available for reads is present. PostgreSQL added a feature to turn this “ongoing” recovery into a read- only access instance, so read queries can be executed even when the WAL apply is active. To activate this feature, the parameter **hot\_standby** needs to be set to **on** in the **postgresql.conf** configuration file at the replica side.

Once the **hot\_standby** parameter is on, the instance will accept read-only queries after reaching a consistent state. During a standby startup, there will be some time when the clients can not connect to the instance. In the database logs, you will see something similar to the next:

LOG: entering standby mode

... then some time later ...

LOG: consistent recovery state reached

LOG: database system is ready to accept read-only connections

Keep in mind that the operations in both, the primary and the standby, can run into some conflicts. This happens because the data changes on the primary should be re-applied in the standby, but the queries on both can be different. As we saw in *Chapter 6, PostgreSQL Internals*, the MVCC, and the VACUUM are mechanisms used in PostgreSQL to handle the concurrency and perform the cleaning from the *dead tuples*. These operations happen on the primary side and need to be re-applied in the standby, but at some point, a query in the standby might still be reading data that was already deleted from the primary; this situation will prevent the WAL application from continuing.

The WAL application can wait for a while to let the read query on the standby to complete before re-applying the data deletion, but cannot wait indefinitely. To handle this, PostgreSQL supports two parameters: **max\_standby\_archive\_delay** and **max\_ standby\_streaming\_delay**. These can control how long the WAL application should wait to let the read query complete. If any query preventing the WAL application from continuing reaches these thresholds, it will be terminated, and the WAL application will resume. This way, the replication stays close to the changes from the primary.

### Archive recovery

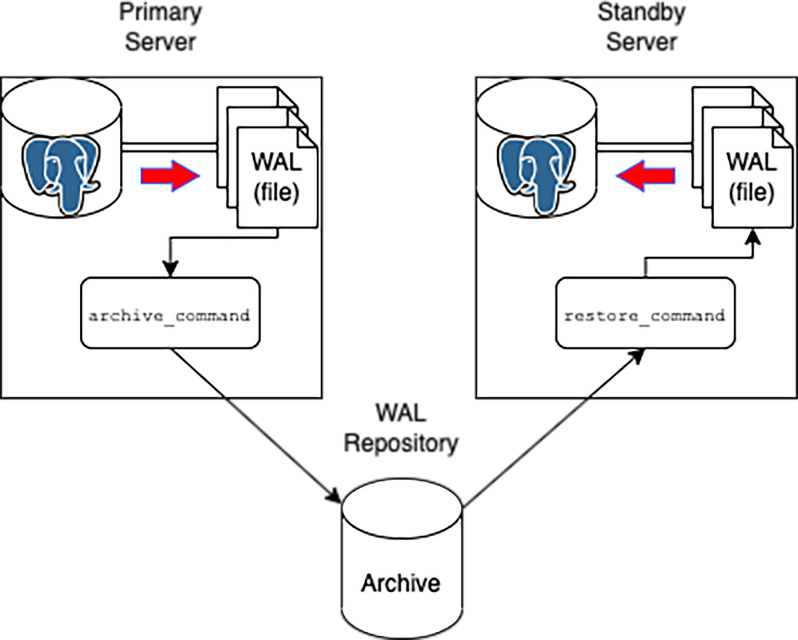
At this point, we have discussed that physical replication is possible by re-applying the data changes from the WAL from the primary into the instance running as a standby in a remote server. The initial way to do this was to physically copy the WAL files from the primary to the standby; then, the remote instance will take them and re-apply the changes.

In *Chapter 7, Backup & Restore in PostgreSQL,* we reviewed the **archive\_command** parameter, which can contain an Operating System command or a script call to execute a routine to archive the WAL files before they get removed or recycled. This usually means copying the WAL from the postgres **data\_directory** to a dedicated repository called *the archive*.

Well, we can build the standby to get the WAL from the same location, *the archive*, where the primary is archiving them. For this, when configuring the standby, we can set the **restore\_command** parameter, which works analogous to the **archive\_ command**, so it can contain an Operating System command or a script call with the routine for getting the WAL from the archive. The next is an example of how this would look:

restore\_command = ‘cp /mnt/server/archivedir/%f “%p”’ restore\_command = ‘copy “C:\\server\\archivedir\\%f” “%p”’ *#Windows*

*Figure 8.3* details these components for a physical replication by archive recovery.



***Figure 8.3:*** *Archive recovery physical replication.*

This physical replication setup has some advantages, such as it is very flexible, you can easily accommodate the primary and the standby server or servers in different physical locations, and they don’t need to “see” each other, having access to the WAL repository is all that they need. But, also has the disadvantage the replication can get too far behind (lag) because of network latency or file transmission issues.

The next is an example of the required configuration to stablish the archive recover

physical replication.

# Primary

wal\_level = ‘replica’ archive\_mode = ‘on’

archive\_command = ‘test ! -f /mnt/server/archivedir/%f && cp %p /mnt/ server/archivedir/%f’

# Standby

hot\_standby = ‘on’

restore\_command = ‘cp /mnt/server/archivedir/%f “%p”’

### Streaming replication

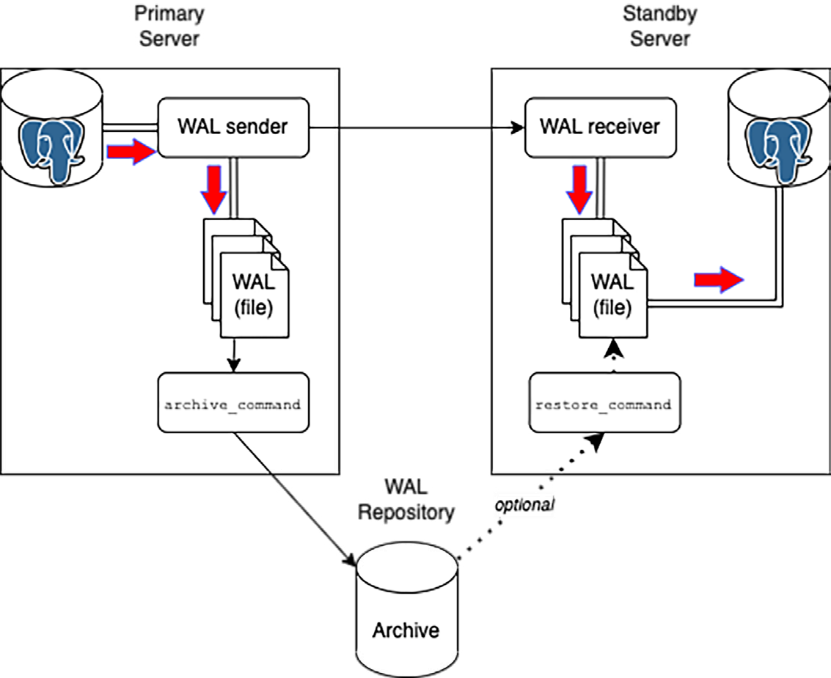
In some way, the streaming replication is an improvement from the archive recovery setup. The foundations of the replication mechanism are the same; the WAL changes from the primary will be re-applied in the standby.

The big difference is now there is no need to copy the WAL files from one server to the other, but a couple of special processes will take care of streaming the WAL changes directly from the WAL buffer in the primary to the remote process in the standby so it can be directly re-applied. The processes are known as **WAL sender** in the primary and **WAL receiver** in the standby.

The configuration of this setup is very similar to the archive recovery; the difference is we need to set the **primary\_conninfo** parameter with the details for the **WAL receiver** to connect to the primary server. The next is an example of how this parameter might look:

primary\_conninfo = ‘host=187.168.1.50 port=5432 user=someuser password=passwd application\_name=s1’

Also, the **restore\_command** is now optional but still useful if the replication stops for a long period, and to resume, needs to get WAL files already archived from the primary. *Figure 8.4* shows this setup.



***Figure 8.4:*** *Streaming replication.*

The streaming replication can keep a nearly real-time sync compared to the archive recovery replication. The direct communication between the **WAL sender** and the **WAL receiver** process saves a lot of time and processing resources. This setup requires proper network configuration to allow communication between the primary and the standby servers.

The next is an example of the configuration parameters to set the streaming

replication.

*# Primary*

wal\_level = ‘replica’ archive\_mode = ‘on’

archive\_command = ‘test ! -f /mnt/server/archivedir/%f && cp %p /mnt/ server/archivedir/%f’ *# optional*

*# Standby*

hot\_standby = ‘on’

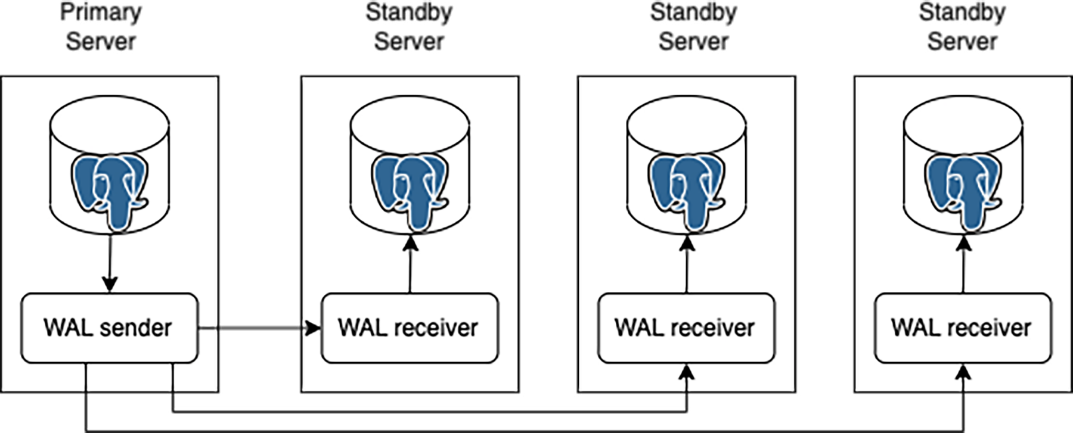
primary\_conninfo = ‘host=187.168.1.50 port=5432 user=someuser password=passwd application\_name=s1’

restore\_command = ‘cp /mnt/server/archivedir/%f “%p”’ *# optional*

### Cascading

Usually, when working with streaming replication, we can take advantage of the almost instant sync status between the primary and the standby servers. Our application design can consider driving the reading load directly to one or more standby; this way, the primary server can fully use its hardware resources to server the read-write operations for the data changes and save the resources from the read- only operations.

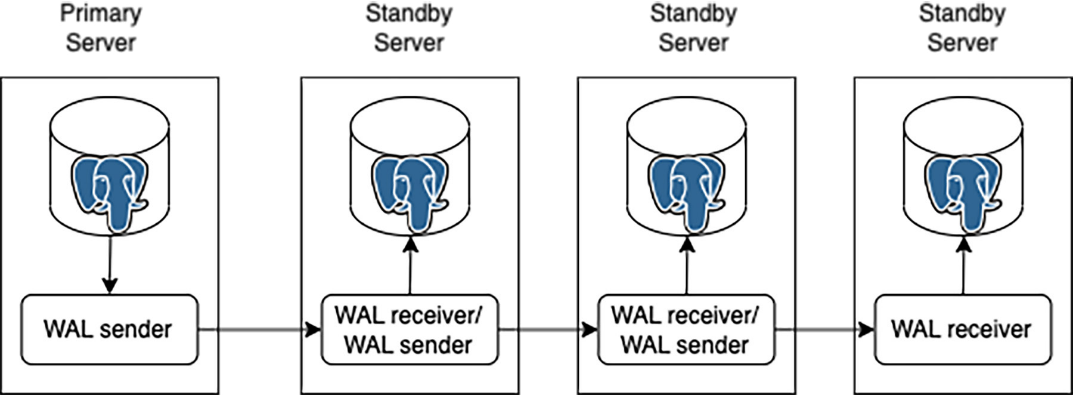
However, in large environments, including multiple standby servers, the primary can suffer from the extra load of many WAL receiver processes. *Figure 8.5* illustrates this setup with multiple standby servers replicating from a single primary server.



***Figure 8.5:*** *Multiple standby physical replication.*

To prevent overwhelming the primary with the load from many standby nodes, we can use another PostgreSQL feature for the physical replication: cascading replication. When using the cascading replication setup, all the previous concepts remain valid; the difference is one or more standby nodes can serve as the upstream for another standby node, preventing the primary from getting this extra load. *Figure*

*8.6* shows how this appears:



***Figure 8.6:*** *Cascading physical replication.*

Configuring the cascading replication requires setting the **primary\_conninfo** in the downstream server, pointing to another standby server that will serve as the upstream.

The following lines show an example of the configuration to achieve this replication

type.

*# Primary*

wal\_level = ‘replica’

archive\_mode = ‘on’

archive\_command = ‘test ! -f /mnt/server/archivedir/%f && cp %p /mnt/ server/archivedir/%f’ *# optional*

*# Standby*

hot\_standby = ‘on’

primary\_conninfo = ‘host=<the primary or another standby as upstream> port=5432 user=someuser password=passwd application\_name=s1’

recovery\_target\_timeline = ‘latest’

restore\_command = ‘cp /mnt/server/archivedir/%f “%p”’ *# optional*

***Note:* For the cascading replication setup is advisable to set the parameter recovery\_target\_timeline to ‘latest’ in the downstream standby nodes, so if the upstream standby gets promoted (it turns into a read-write), the cascading won’t get broken.**

### Delayed replica

Another interesting feature supported by PostgreSQL regarding physical replication is what we know as a **delayed replica**. This method relies on top regular streaming or archive recovery replication. The difference with the delayed replica is that the data changes will be *visible* after a defined time. For example, we plan to have a replication delayed by *X* time so that the data would be accessible in that replica as follows:

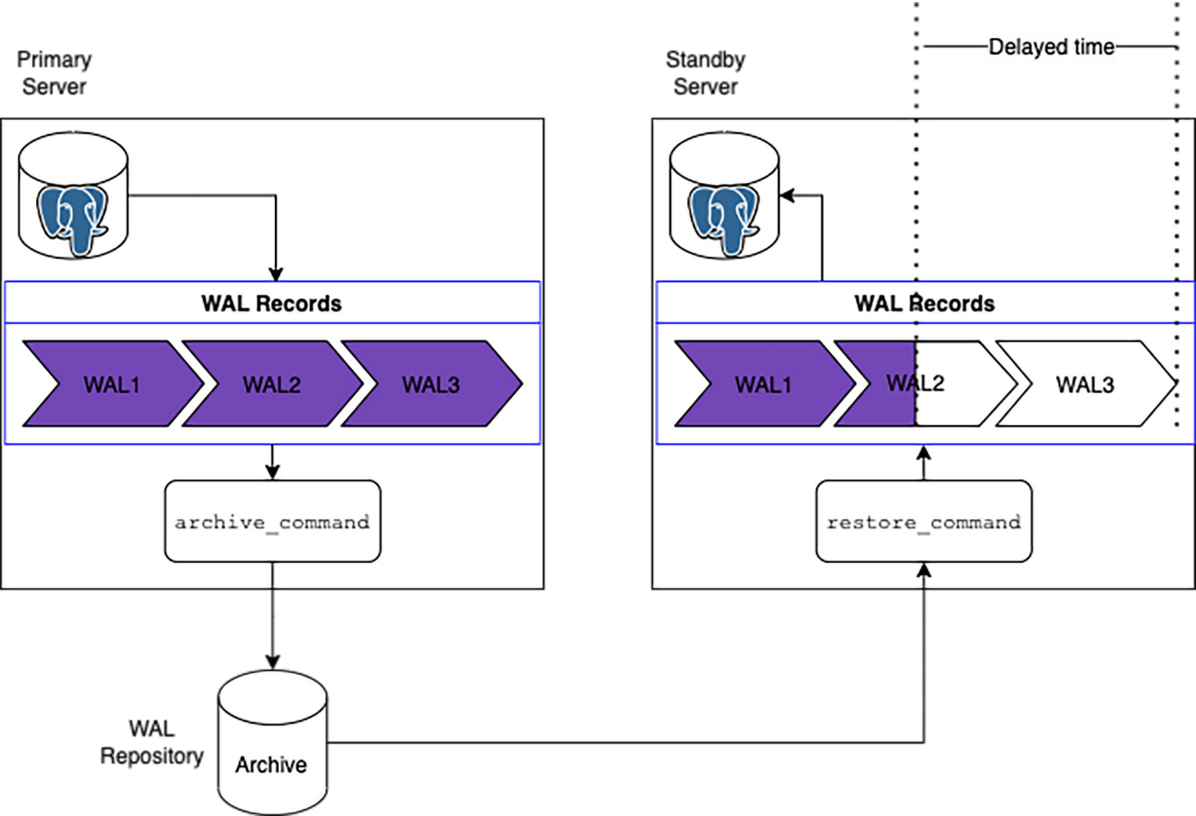
*delayed replica visibility time = data change time + X*

As an example, consider the next:

* We configured a replica delayed by 2 hours.
* There is a data change in the master node applied at 12:22 PM. When will the data change be visible in the delayed replica?

*delayed replica visibility time = 12:22 PM + 2 hours delayed replica visibility time = 2:22 PM*

*Figure 8.7* illustrates this functionality:



***Figure 8.7:*** *Delayed replica*

We need to consider the time calculated regarding the local system time. So, for a practical configuration, we would need to ensure that both the primary and the standby servers are configured equally for their local timing.

As was stated before, the delayed replication is a variation of the *regular* streaming or archive recovery methods. In both cases, the changes are taken from the WAL and applied on the standby. But we can tell that the archive recovery is the more effortless/straightforward way to configure it. So, it will continuously retrieve the WAL from the archive and apply the changes just when the delayed time has passed.

To get this functionality working on a PostgreSQL standby instance, we need to configure the **recovery\_min\_apply\_delay** parameter according to the time we want to keep it delayed from the master. The next is an example of the parameter configuration for the setup of the delayed replica.

*# Primary*

wal\_level = ‘replica’

archive\_mode = ‘on’

archive\_command = ‘test ! -f /mnt/server/archivedir/%f && cp %p /mnt/ server/archivedir/%f’ *# optional*

*# Standby*

hot\_standby = ‘on’

primary\_conninfo = ‘host=187.168.1.50 port=5432 user=someuser password=passwd application\_name=s1’

recovery\_min\_apply\_delay = ‘2h’

restore\_command = ‘cp /mnt/server/archivedir/%f “%p”’ *# optional*

We can use a delayed replica to provide a quick way to recover data or schema definition in the case of an erroneous deletion, table truncation, drop, or failed application deployment, which has changed a large piece of data making the undo complicated.

### Configuration

We have reviewed the different features and topologies we can get for the physical replication in PostgreSQL. The next is a summary of the configuration parameters example we need to keep in mind to build the same.

When working with any physical replication setup, we need to “tell” the standby nodes they are supposed to be a replica, starting with PostgreSQL 12, and onwards this is done by adding an empty file in the **data\_direcotory** (PGDATA) called **standby.signal**. With this file in place, once the postgres server is started, it will enter the replica mode, depending on the parameters we have configured.

## Logical replication

Another type of replication is logical replication, where data is replicated logically instead of physically replicating block by block at the file system level. It is the same as taking logical backup, which we have seen in detail in *Chapter 7, Backup & Restore in PostgreSQL***.**

### Architecture

Logical replication uses a publisher and subscription model where the publisher pushes the data to one or more subscribers. The subscriber pulls the data from the publishers to which it is subscribed. One can configure as complex an architecture as possible using publisher and subscriber models, like one publisher and one subscriber, one publisher and multiple subscribers, cascading replication, and many more, as shown in the previous sections of this chapter.

Tables whose data needs to be replicated must have a replica identity, usually the Primary/Unique Key. It starts by initially taking a snapshot of the tables in the publisher node. The subscriber will initially pull those changes from the publisher, which takes time, depending on the table size.

Any transactional changes are replicated as and when the transaction occurs. The subscriber applies the changes in the same order in which they happen on the publisher node. The main thing to remember while setting up the logical replication is that it can only replicate **Data Manipulation Language** (**DML**) changes such as **INSERT**, **DELETE**, or **UPDATE**, not **Data Definition Language** (**DDL**) changes like **CREATE**, **ALTER**, or **DROP**.

The replication will break in case any changes to table structure in the form of altering the table are applied to the tables on which replication has been set. Also, the subscriber node may or may not be the read-only node. The only thing to be kept in mind is to avoid conflicts and refrain from performing write operations on the same set of tables on which logical replication has been configured.

### Publication

The publication can be described same as the master node in the physical replication. The node on which the publication is configured is called the publisher node, which is responsible for sending data to the subscription node. The process which is used to send the data is WAL SENDER.

The publisher can be created using the **CREATE PUBLICATION** command and modified or dropped using **ALTER PUBLICATION 1** Command. The tables used for logical replication must have replication identity in the form of UNIQUE/PRIMARY KEY. Data is replicated in real-time once the initial snapshot of the tables is created. Publisher can choose to replicate the changes for either INSERT, UPDATE, **DELETE** or **TRUNCATE**, or a combination of any of them.

Syntax:

**CREATE PUBLICATION name**

**[ FOR TABLE [ ONLY ] table\_name [ \* ] [, ...]**

**| FOR ALL TABLES ]**

**[ WITH ( publication\_parameter [= value] [, ... ] ) ]**

### Subscription

Subscription is like a standby node in the physical replication. The node which pulls data from the publisher is called the subscriber node. Subscription can be added on the child node using the **CREATE SUBSCRIPTION** command and modified or deleted using **ALTER SUBSCRIPTION** command.

Syntax:

**CREATE SUBSCRIPTION subscription\_name CONNECTION ‘conninfo’**

**PUBLICATION publication\_name [, ...]**

**[ WITH ( subscription\_parameter [= value] [, ... ] ) ]**

**Configuring the Logical Replication:**

Once tables have been identified to be set for logical replication, make the configuration changes as following:

Changes in **postgresql.conf** file:

**Publisher Node:**

*wal\_level = logical*

### Publisher node as well as subscription node

Apart from this, in case replication slots have been used, then set **max\_replication\_ slots** same as the number of subscriptions which will be connecting. Also, add some more as a buffer which will be used for table synchronization. **Max\_wal\_senders** should be set as **max\_replication\_slots** .Additionally **max\_worker\_processes** can be set as (**max\_replication\_slots + 1**). Except **wal\_level**, all other parameters will be same at the publisher and subscriber nodes.

1. Changes in **pg\_hba.conf** file:

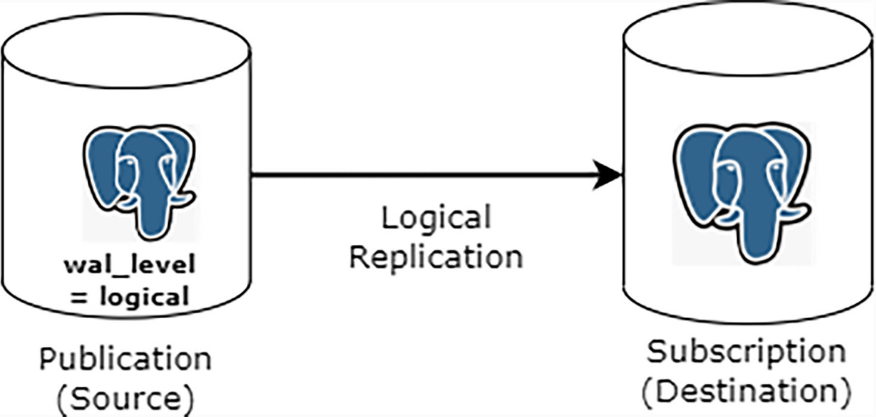
Allow replication user in **pg\_hba.conf** file for both the nodes:

*host all replication\_user 192.168.80.2/32 md5*

1. Create Publication on the Publisher node as per the following example:

*CREATE PUBLICATION pub\_test FOR TABLE emp, dept;*

1. Create Subscription on the Subscriber node as per the following example: *CREATE SUBSCRIPTION sub\_test CONNECTION ‘dbname=db1 host=192.168.80.1/32 user=replication\_user’ PUBLICATION pub\_test;*



***Figure 8.8:*** *PostgreSQL Logical Replication*

Please find the other use cases of logical replication as mentioned following:

* + Logical replication can be used to replicate data between different versions of PostgreSQL. It is majorly used for performing data migration while upgrading the database.
  + Since it is not managed at the OS or file system level, it works very nicely for performing replication with different platforms. For example, replicating data from Windows to Linux or replicating data amongst different flavors of Linux/Unix, and so on.
  + Unlike physical replication, where one cannot replicate part of the PostgreSQL database cluster, logical replication can send incremental data changes to a single database or subset of the database. For example, BI & reporting team needs access to specific tables only and wants to perform some operations on its own schema tables; one can setup logical replication for a specific table from production, which will push the incremental changes in the same database where reporting database is hosted. BI Team does not need to have access to the entire production database just to access a few tables.

* + Sharing incremental changes of a specific set of tables to multiple databases.

Amalgamate numerous databases into one database.

Please find the limitations of Logical Replication as following:

* + Tables used in logical replication must have a Primary Key/Unique Key.
  + Only DML can be replicated, and DDL cannot be replicated.
  + TRUNCATE is also not replicated.
  + SEQUENCES and views cannot be replicated.

## Extensions

As we have seen in the past chapters, PostgreSQL is a very robust database software. The base distribution from the community packages already contains unique functionalities that cover many operations out of the box. However, the design of PostgreSQL makes it able to extend its capabilities by integrating what we know as **extensions**.

An extension is a set of the control file, SQL files, and loadable libraries. The libraries are usually written in C language; once they are loaded, PostgreSQL can use the functionalities like any other core function.

Adding an extension to PostgreSQL generally requires two or three steps, depending on whether the extension routines need to perform operations at the server start.

* Install or get the extension libraries at the operating system level. Usually, installing packages accordingly to the Operating System distribution we are using.
* Some extensions require to be added to the **shared\_preloaded\_libraries** parameter in the **postgresql.conf** file. Any change to this parameter requires a PostgreSQL restart to become active. During the startup routine, the added libraries will also be loaded and perform their tasks, such as allocating memory or starting background processes.
* Create the extension within the PostgreSQL database where we will use it. Once the libraries are placed at the operating system level, we can execute the **CREATE EXTENSION** command to load the library and create all the required database objects.

As we saw in *Chapter 2, Getting PostgreSQL to work***,** *Figure 2.3*, when installing PostgreSQL from the source code, we get a folder called **contrib/**, which is shipped with the source and contains several extensions. The PostgreSQL core community maintains all these extensions; however, there are others developed externally.

In the following subsections, we will review some extensions widely used and helpful for various projects. We will study the steps to add them to PostgreSQL version 14 running on Ubuntu and the basics for their usage.

### pg\_cron

This extension lets to schedule database jobs at specific times; it uses the same time and job conventions as the standard **cron** utility for Linux operating systems. So you can easily schedule recurrent database tasks or maintenance events. The following are some examples.

*-- Execute cleaning data custom function on Sunday at 3:00 AM GMT* SELECT cron.schedule(‘0 3 \* \* 7’, $$SELECT f\_clean\_data()$$);

schedule



8

*-- Runs vacuum daily at 5:30 AM GMT*

SELECT cron.schedule(‘daily-vacuum’, ‘30 5 \* \* \*’, ‘VACUUM’); schedule



9

*-- Stop scheduling daily-vacuum job*

SELECT cron.unschedule(‘daily-vacuum’ ); unschedule



t

The Citus Data company currently supports this extension as an open-source project

*(Reference: pg\_cron).*

Consider running a PostgreSQL version 14 on Ubuntu operating system. The

following are the steps to integrate and use this extension.

1. Install the required package for the PostgreSQL version on Ubuntu.

sudo apt-get -y install postgresql-14-cron

1. The **pg\_cron** extension requires to be pre-loaded at server startup so that it can allocate background workers. So, we need to add the next to the **postgresql.conf** file and restart postgres.

*# edit postgresql.conf*

shared\_preload\_libraries = ‘pg\_cron’

1. By default, the extension expects to be added to the **postgres** database and to schedule the jobs based on the GMT timezone. You can modify these defaults to use a different database and another timezone, for example, database *prod1* and the timezone *CST*.

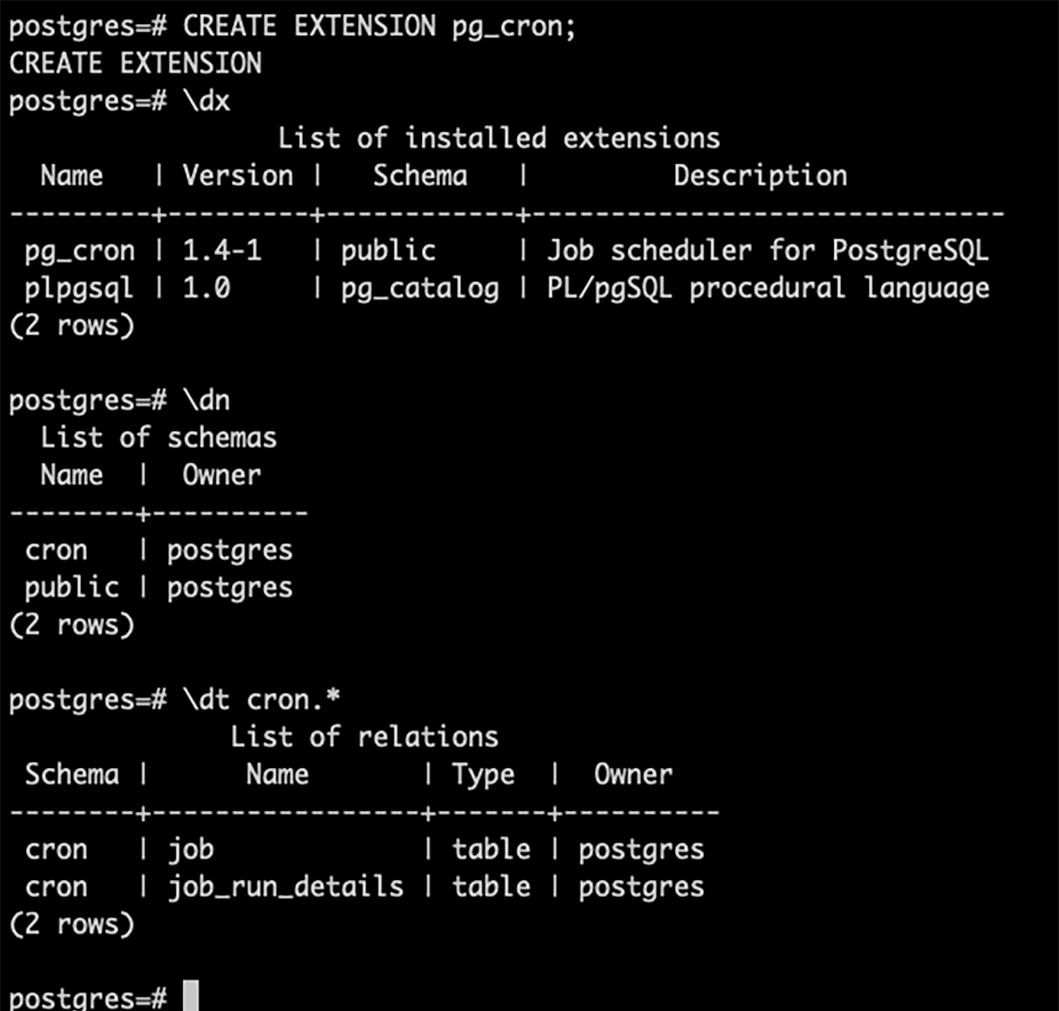
*# edit postgresql.conf* cron.database\_name = ‘prod1’ cron.timezone = ‘CST’

1. Finally, once you have added the previous configuration and restarted

PostgreSQL, you need to create the extension in the desired database. The following should be executed as a superuser.

CREATE EXTENSION pg\_cron;

*Figure 10.1* illustrates the creation of the extension within the **postgres** database and shows the created schema called **cron** and its tables.



### pg\_stat\_statements

This is a really interesting and valuable extension. Once loaded and added to a database, this extension will collect statistics about the executed queries, such as the query text, the query identifier, the user and database names, the number of times the query executes, the total time spent in the planning phase and the time spent in the execution phase, the number of rows retrieved, and much more.

All this information comes in handy when tracking performance issues or verifying what queries are causing the most load in the system so you can plan improvements. You can access the collected data through a database view named the same as the extension: **pg\_stat\_statements**. The following are some examples:

*-- Find the query with the highest total execution time* SELECT max(total\_exec\_time) AS max\_exec\_time, query FROM pg\_stat\_statements

GROUP BY query

ORDER BY max\_exec\_time DESC LIMIT 1 ;

max\_exec\_time | query

*--------------------+-----------------------------------------------*



216302.15497300148 | UPDATE pgbench\_branches SET bbalance = bbalance

+ $1 WHERE bid = $2 (1 row)

*-- Find the query executed the most times* SELECT max(calls) AS max\_calls, query FROM pg\_stat\_statements

GROUP BY query

ORDER BY max\_calls DESC LIMIT 1 ;

max\_calls | query

*-----------+--------------------------------------------------------*



72981 | UPDATE pgbench\_accounts SET abalance = abalance + $1 WHERE aid = $2

(1 row)

*-- Find the query which retrieved the most rows*

SELECT max(rows) AS max\_rows, query FROM pg\_stat\_statements

GROUP BY query

ORDER BY max\_rows DESC LIMIT 1 ; max\_rows | query

*----------+----------------------------------*

100000 | copy pgbench\_accounts from stdin (1 row)

*-- Find the query with the worst cache hit ratio (most reads from disk)*

SELECT min(100.0 \* shared\_blks\_hit/nullif(shared\_blks\_hit + shared\_ blks\_read, 0)) hitratio,

substr(query,0,50) query FROM pg\_stat\_statements

GROUP BY query

ORDER BY hitratio ASC NULLS LAST LIMIT 1 ;

hitratio | query

*---------------------+----------------------------------------------*



84.6153846153846154 | SELECT e.extname AS “Name”, e.extversion AS “Vers

(1 row)

This extension is currently delivered as part of the **contrib/ package** from the base community distribution *(Reference: pg\_stat\_statements).*

The following are the steps to add the extension to PostgreSQL.

1. Considering this example is running on Ubuntu, when installing the postgresql server package, the **postgresql-contrib** package also gets installed. This last one contains multiple extensions, as we saw above. So the extension libraries are already present.
2. Due to the fact this extension requires allocating some extra shared memory at the postgres startup, you need to add it to the **shared\_preloaded\_libraries** parameter.

*# edit postgresql.conf*

shared\_preload\_libraries = ‘pg\_stat\_statements’

1. The extension supports a few configuration parameters, so optionally, you can adjust them in the **postgresql.conf** file. For example, you can change the number of queries to track from the default 5000 to 10000. Check the bibliography entry cited above for extra details about the configuration parameters.

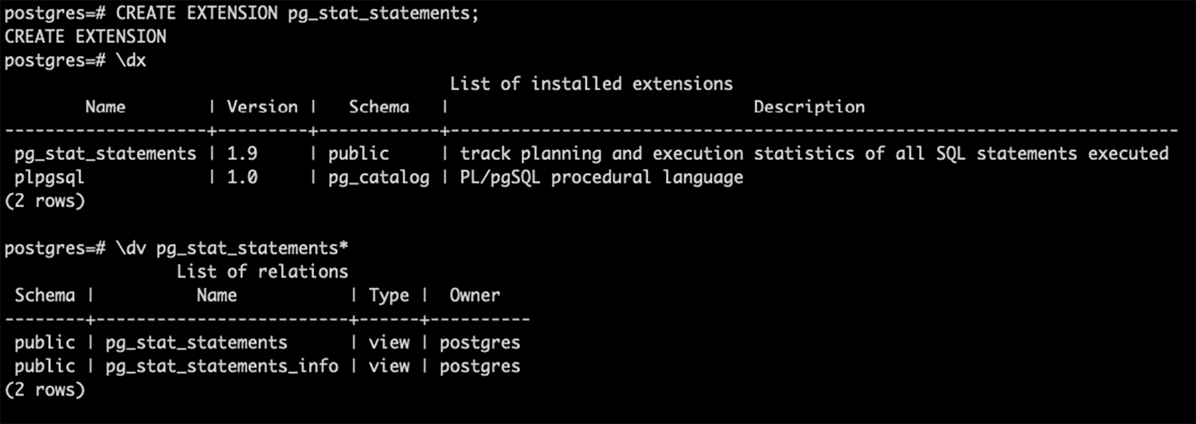
*# edit postgresql.conf*

pg\_stat\_statements.max = 10000

1. After adding the configuration and restarting postgres, you can create the extension in the database.

CREATE EXTENSION pg\_stat\_statements;

In *Figure 10.2*, you can see the result of creating the extension; it will add new views:



***Figure 10.2:*** *pg\_stat\_statments extension*

### pg\_repack

We will not lie if we say this is one of the most potent and helpful extensions for routine DBA tasks. As we saw in previous chapters, PostgreSQL uses **MultiVersion Concurrency Control** (**MVCC**) to handle multiple concurrent user sessions and maintain data integrity and consistency.

Over time, the **vacuum** removes the dead tuples generated because of the different row versions, but it does not return this space to the operating system. In certain circumstances, this *unused* space within the tables or indexes pages gets large, and even when the database might use it to accommodate new rows, sometimes it stays unused; this is what we know as bloat, negatively impacting data access performance.

The only way to eliminate bloat is to rebuild the tables or indexes. We can do that with the VACUUM FULL, but it comes with a high cost; the database object gets locked as long as the task remains. Here is where **pg\_repack** comes in.

This extension and its **Command-Line Interface** (**CLI**) counterpart help rebuild database objects with minimal locking so you can securely use them on a running system.

**pg\_repack** works by creating a *shadow* object and copying the data from the source. The original object remains operating with no lock, and all the rebuild happens in a different table or index. In the end, the original object is dropped, and the new one is renamed like the original. The next is an example of bloat in a given table and how it looks after the repack.

Doing a rows count first:

SELECT count(1) FROM pgbench\_accounts;

count



3000000

(1 row)

Calling a SQL script to check the bloated size and percentage *(Reference: Table bloat)*.

\i bloat.sql

schemaname | tblname | real\_size\_pretty | bloat\_size\_byte\_ pretty | bloat\_percentage

*------------+------------------+------------------+-----------------*

*-------+------------------*

public | pgbench\_accounts | 384 MB | 5592 kB | 1.4

(1 row)

Deleting 1 million rows:

DELETE FROM pgbench\_accounts WHERE aid IN (SELECT aid FROM pgbench\_ accounts LIMIT 1000000) ;

DELETE 1000000

Refreshing table statistics: ANALYZE pgbench\_accounts ; ANALYZE

Checking the bloat again:

\i bloat.sql

schemaname | tblname | real\_size\_pretty | bloat\_size\_byte\_ pretty | bloat\_percentage

*------------+------------------+------------------+-----------------*

*-------+------------------*

public | pgbench\_accounts | 384 MB | 132 MB | 34.4

(1 row)

Now we can see 132MB from the total of 384MB is bloat, representing 34.4%

From the operating system prompt, execute the pg\_repack CLI for the given table: postgres@ubuntu-focal:~$ pg\_repack --table=pgbench\_accounts postgres INFO: repacking table “public.pgbench\_accounts”

postgres@ubuntu-focal:~$

After refreshing the statistics again, we can check the bloat after the repack:

postgres=*# \i bloat.sql*

schemaname | tblname | real\_size\_pretty | bloat\_size\_byte\_ pretty | bloat\_percentage

*------------+------------------+------------------+-----------------*

*-------+------------------*

public | pgbench\_accounts | 256 MB | 3720 kB | 1.4

(1 row)

The total table size is now 256MB, and all the unused space got removed. The new

bloat size is 3720KB representing 1.4%.

Finally, verify the table has the expected rows after the DELETE we did before:

postgres=*# SELECT count(1) FROM pgbench\_accounts;*

count



2000000

(1 row)

The **pg\_repack** project started as a fork of another called **pg\_reorg**, but it has gained its reputation. It is maintained and distributed as an open-source development *(Reference: pg\_repack).*

The following are the required steps to add this extension to PostgreSQL.

1. Install the package at the operating system level. In this case, Ubuntu.

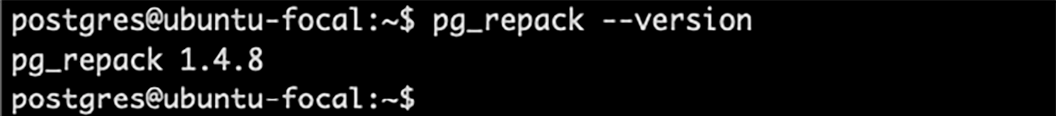
apt install postgresql-14-repack

1. Create the extension in the desired database.

CREATE EXTENSION pg\_repack;

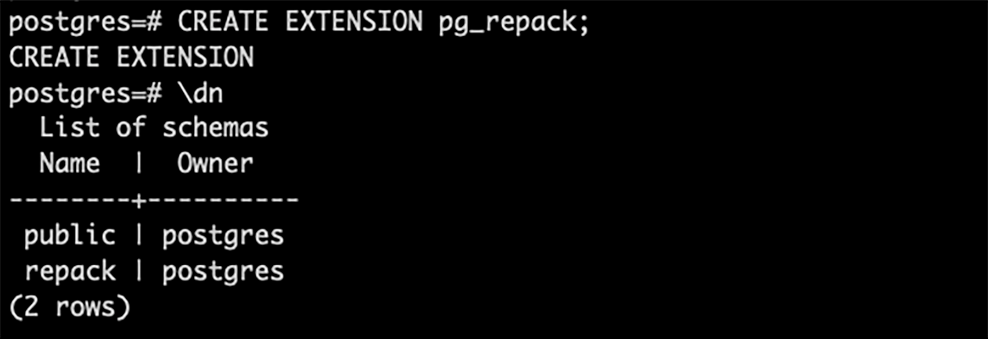
As a result of the two previous commands, you will end with the operating system

CLI utility, and the PostgreSQL extension added. *Figure 10.3* illustrates the CLI utility:



***Figure 10.3:*** *pg\_repack operating system CLI*

*Figure 10.4* shows the extension created in the database and the repack schema added;

this will contain some temporary log tables used to track the repack operations.

***Figure 10.4:*** *pg\_repack extension*

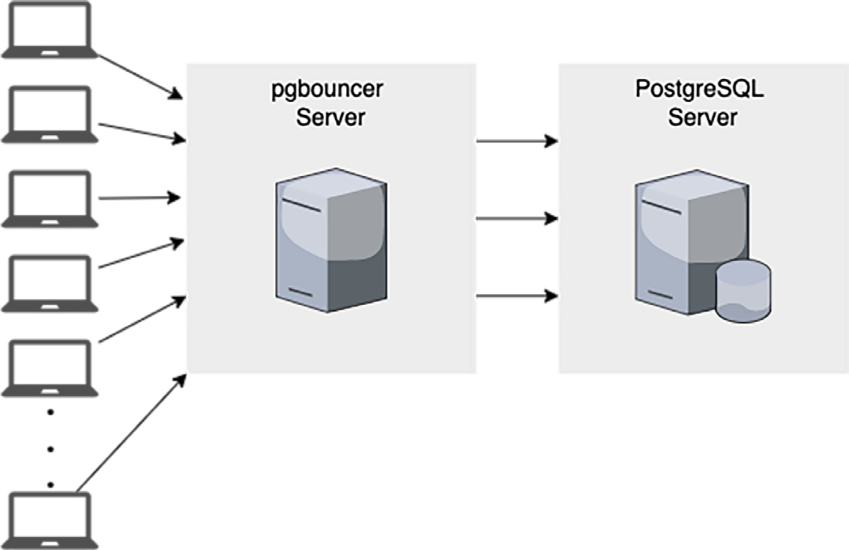
#### pgbouncer

If you remember from previous chapters, PostgreSQL is a process-oriented software that relies on operating system processes for every executable component of its architecture. So, when you start a new PostgreSQL service, you will see many new process-ID (PIDs) running in the operating system, one for the postmaster, another for the checkpointer, another for the autovacuum, and so on. This also applies to the client or user connections; every new user session that connects to the postgres will create a new PID.

This design can work perfectly fine and perform as wonder. But as the number of concurrent sessions rises because of application growth, new application releases, or simply because of an increment in your user base, performance issues due to the high number of running processes in the operating system can show up.

Reducing the number of concurrent processes in the database is the best way to deal with resource saturation at the operating system level, but definitely, we will not want to limit or reduce the number of users that want to be attended by our system. So, to reduce the number of processes in the database end without sacrificing the number of clients we can serve, we can use **pgbouncer**.

The **pgbouncer** tool is a pooler application that can help to keep a large number of client requests being attended to through a reduced number of database connections. *Figure 10.7* illustrates this.



***Figure 10.7:*** *pgbouncer setup.*

**pgbouncer** allows you to configure different pools; depending on the databases you need to connect and the users that will connect, you can define the size of the pool (backend sessions) and the number of client connections to accept. The size of the pools depends on the resource the database server has, mainly CPUs, and the number of connections you need to attend; however, the goal is to set the client connections significantly higher than the number of database connections.

Once installed, **pgbuncer** can be configured by editing the **pgbouncer.ini** file. The following is an example of a basic configuration.

[databases]

pg1 = host=localhost dbname=pg1 auth\_user=appuser pool\_size = 20

[pgbouncer] pool\_mode = session listen\_port = 6432

listen\_addr = localhost auth\_type = md5 auth\_file = users.txt logfile = pgbouncer.log pidfile = pgbouncer.pid admin\_users = adminuser

stats\_users = stat\_collector max\_client\_conn = 200

In the above configuration file example, the next definitions are taking place:

* The database connection called **pg1** will create a pool of 20 backend sessions.
* The pool will attend the client sessions in **session** mode, which means a backend session from the pool will stick to a client session until it finishes; after that, the backend session can be assigned to a different client session.
* The maximum number of client connections this pgbouncer will attend is

200.

The **pgbouncer** tool is maintained by its own community (Reference: pgbouncer) and is available as an open-source project. The next are the steps to get this tool.

1. You just need to install the proper package for your operating system; as with other open-source software, you may compile it by yourself. In the case of Ubuntu, you can get it with the next.

apt install pgbouncer

***Performance Tuning***

## Introduction

Any **Relational Database Management System** (**RDBMS**) intends to resolve user queries as quickly as possible, and PostgreSQL is no exception. So, keeping postgres well performant is one of the main goals for **Database Administrators** (**DBAs**).

In previous chapters, we have studied many of the capabilities of PostgreSQL and how to configure and use them. Now, we will dive into the performance tuning topics. We will learn about the basic concepts and components involved in database tuning and some best practices we can use as an initial guide.

## Structure

During this chapter, we will study the next sections:

* Indexes
* Statistics
* Explain plan
* Best practices for the **postgresql.conf** parameters

## Objectives

Once you complete this chapter, you will be familiar with the basic components for tuning in PostgreSQL and understand their relationship and impact. Also, you will relate the standard best practices when installing or tuning a postgres system to deliver the best performance.

## Indexes

Have you ever noticed the book indexes that help locate the chapters by mentioning the page number of the chapter? One can jump to that page directly rather than going through each chapter sequentially which takes a lot of time. In the same way, indexes also create a similar kind of metadata like page numbers which helps to jump to the requested data directly instead of scanning the whole table sequentially.

This benefit comes with an overhead since adding an index involves it being created/ modified as data gets inserted, updated, or deleted from the table. This is one of the reasons one should create wisely through proper analysis. If the table is used for read-only operations more than the write operations, then it could be beneficial to create the indexes; else, it might become an overhead to update the index on every write operation.

Syntax as per PostgreSQL Documentation: *(Reference - PostgreSQL Community Index)*

CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] ***name*** ] ON

[ ONLY ] ***table\_name*** [ USING ***method*** ]

( { ***column\_name*** | ( ***expression*** ) } [ COLLATE ***collation*** ] [ ***opclass***

[ ( ***opclass\_parameter*** = ***value*** [, ... ] ) ] ] [ ASC | DESC ] [ NULLS

{ FIRST | LAST } ] [, ...] )

[ INCLUDE ( ***column\_name*** [, ...] ) ] [ NULLS [ NOT ] DISTINCT ]

[ WITH ( ***storage\_parameter*** [= ***value***] [, ... ] ) ] [ TABLESPACE ***tablespace\_name*** ]

[ WHERE ***predicate*** ]

Generic Syntax normally used:

CREATE INDEX <schema\_name>.<index\_name> on <table\_name> (<column/ column-list>)

CREATEing and DROPping the index is not an online activity by default. The table is locked for any modification. However, read operations can be performed while the index is being created/modified by including the **CONCURRENTLY** keyword in the index creation statement. It will help to modify indexes concurrently online, as shown in the syntax as per PostgreSQL documentation above.

Indexes on multiple columns can also be created, which are called Multi Column Indexes. While creating such indexes, make sure that the order of columns is the same as used in the query’s **WHERE** clause.

### Reindex

Reindexing an index is one of the maintenance activities in PostgreSQL, which helps to rebuild the corrupted/bloated/invalid index concurrently. An index can also be rebuilt when storage is changed for the specific index.

Syntax as per PostgreSQL Documentation: *(Reference - PostgreSQL Community Re Index):*

REINDEX [ ( ***option*** [, ...] ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } [ CONCURRENTLY ] ***name***

where ***option*** can be one of:

CONCURRENTLY [ ***boolean*** ] – Reindex can be created concurrently online

TABLESPACE ***new\_tablespace – Reindexing could be done in new tablespace***

VERBOSE [ ***boolean*** ] – Logs will be printed while reindexing

Whenever a Primary Key or Unique Key constraints are created, by default the system creates indexes on such columns as commonly those columns will be extensively used for fetching the data in the queries. Generally, the indexes could be created on columns used to filter data, that is, WHERE clause columns. It is not a thump rule; however, it is a good practice to create indexes on those columns that are used as the child table’s foreign key. Typically such columns will be used in join conditions to fetch the data.

Indexes might not be needed for small databases initially, as default indexes created on Primary Key or Unique are enough. However, as time goes on, data also increases, and it takes more time than it usually takes. One can use **EXPLAIN PLAN** to find out

whether indexes are used by the query to fetch the data. **EXPLAIN PLAN** is discussed in detail later in this chapter.

Indexes are one of the building blocks to improve the performance of the queries. These queries could be the reporting queries that fetch data or any **SELECT** queries that need improvement in the query execution time.

Indexes might also make the performance of **INSERT, UPDATE**, and **DELETE** queries in which a **WHERE** clause is used for search conditions. The **ANALYZE** command is recommended so that statistics of the indexes can be kept up to date, and whenever it is used, it will contain updated metadata which will help to enhance performance better.

### Index types

Each query is different and may not give the best results with the default BTREE index. To overcome this, PostgreSQL has different types of indexes, which help increase the performance of various kinds of queries. If one wants to create a specific index type apart from the default BTREE index, it can be done with the **USING** keyword in the index creation statement. Let us now understand the different types of indexes in PostgreSQL.

#### Btree index

As discussed earlier, by default BTREE index is created when no index type is mentioned in the **CREATE INDEX** statement. The b-tree index sorts the data and keeps it sequentially. It works well when the *equal-to* operator is used in the comparison in the query’s **WHERE** clause. Along with *equal-to* BTREE works well with the below operators as well:

* <
* >
* <=
* >=
* IS NULL
* IS NOT NULL
* BETWEEN
* IN

B-tree indexes can also be used to retrieve data in sorted order. This is not always

faster than a simple scan and sort, but it is often helpful.

#### Hash index

Hash Index generates a 32-bit hash code on the column where the index is created. It works best when the *equal* operator is used in the where clause of the query.

#### GiST and SP-GiST index

These indexes are used when two-dimensional geometrical data types are used.

#### Gin index

This index is used when one-dimensional data types like ARRAYS are used in the data types.

#### Brin index

**BRIN** indexes (a shorthand for **Block Range INdexes**) store summaries about the values stored in consecutive physical block ranges of a table. Thus, they are most effective for columns whose values are well-correlated with the physical order of the table rows. *(Reference: PostgreSQL Community Index Types).*

### Indexes and expressions

An index column need not be just a column of the underlying table but can be a function or scalar expression computed from one or more columns of the table.

For example,

—-WHERE clause with expression

SELECT \* FROM people WHERE (first\_name || ‘ ‘ || last\_name) = ‘John Smith’;

—-INDEX creation with expression same as WHERE clause

CREATE INDEX people\_names ON people ((first\_name || ‘ ‘ || last\_name));

The system sees the query as just *WHERE indexed-column = ‘constant’* and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

The metadata views - **pg\_index** and **pg\_indexes** give information about the details

of the indexes like the schema name, index name, and the like. We will see more examples of INDEX later in this chapter.

## Statistics

Statistics are what we usually call metadata, meaning *the data about the data*. As we saw in the *Query Processing* section of *Chapter 6*, the planner is the stage in charge of determining the fastest and cheapest path to get the data required by a query. To accomplish that, it relies on the collected statistics.

There are two main types of statistics in PostgreSQL, each maintained by different

components.

* The statistics about the server activity. These are collected and maintained by the stats collector background process; it feeds a set of catalog views with information about the usage of the system objects, such as the count of accesses to the tables and indexes or the number and state of the user sessions. Usually, this information is used by the Database Administrator to verify the system’s state.
* The statistics about the data distribution within the tables and indexes. This information is collected when the **ANALYZE**, or **VACUUM ANALYZE** commands are executed manually or by the autovacuum background process. This is the information the planner uses, and we will focus on this section.

The statistical information about the tables and indexes data is stored in three different system catalogs, depending on their kind:

* **pg\_class**: In this catalog, the system stores statistics about the number of tuples (rows) in the tables or indexes and the occupied blocks in the disk, alongside some objects’ identity details.
* **pg\_statistics**: Here postgres stores information about the *selectivity* of the data. It has one or two rows per table column, depending if the table has child tables (inheritance). This information might be hard to read since it is intended to be used by the system itself; however, postgres has a view called **pg\_stats** which shows the same information in a more easy-to-understand way.
* **pg\_statistics\_ext\_data**: Here additional information is stored in the case we add an *extended statistics object* for a specific set of tables columns.

In the following paragraphs, we will study each of these in detail and see a few

examples.

To illustrate the different statistics types, we will use an example database called **pagila**, a port from the MySQL **sakila** example database. *(Reference: Devrim Gündüz pagila).*

Statistics in **pg\_class**

Considering the tables and data from the example database **pagila** we can review the statistics in the **pg\_class** catalog. For example, consulting the information for the address table and its index:

pagila=# SELECT

relname AS “relation\_name”, relkind AS “relation\_kind”, reltuples AS “relation\_tuples”, relpages AS “relation\_pages”,

pg\_size\_pretty(pg\_relation\_size(oid)) AS “relation\_size” FROM pg\_class

WHERE relname LIKE ‘address%’ AND relkind IN (‘r’, ‘i’);

-[ RECORD 1 ]---+-------------

relation\_name | address relation\_kind | r relation\_tuples | 603

relation\_pages | 8 relation\_size | 64 kB

-[ RECORD 2 ]---+-------------

relation\_name | address\_pkey relation\_kind | i

relation\_tuples | 603

relation\_pages | 4 relation\_size | 32 kB

This catalog shows that the table and the index have 603 rows. The table uses 8

pages, each of 8 KB occupying 64 KB in total in the disk. And, how we could guess the index is smaller and uses only 4 pages with a total of 32 KB. The planner uses this information to know how much work it would take to read from these objects.

The number of tuples is not updated in real time. As we saw before, the statistics are updated by the **ANALYZE** or **VACUUM ANALYZE** commands, but the planner uses the information about the used pages to escalate the estimation of the total number of rows, so the approximation is closer to the actual number.

### Statistics in pg\_statistics

The information in the **pg\_class** is OK when the queries aim to read the complete table data, so the planner can know the total number of rows it will retrieve. However, the most common operations are executed to retrieve just a subset of rows from the table.

To improve these filtering operations, using the **WHERE** clause, postgres keeps statistics about the selectivity of the values contained per table column. This information is stored in the **pg\_statistics** catalog, and as we saw above, there is a more human- readable view called **pg\_stats**. Considering the same sample database, we could verify the information of the **postal\_code** column from the **address** table:

pagila=*# SELECT*

attname AS “column\_name”, n\_distinct AS “distinct\_rate”,

array\_to\_string(most\_common\_vals, E’\n’) AS “most\_common\_values”,

array\_to\_string(most\_common\_freqs, E’\n’) AS “most\_common\_ frequencies”

FROM pg\_stats

WHERE tablename = ‘address’ AND attname = ‘postal\_code’;

-[ RECORD 1 ]-----------+-------------

column\_name | postal\_code

distinct\_rate | -0.9900498 most\_common\_values | +

| 22474 +

| 52137 +

| 9668

most\_common\_frequencies | 0.006633499 +

| 0.0033167496+

| 0.0033167496+

| 0.0033167496

The **pg\_stats** view has some other details, but in this example, we can see the rate of distinct values, the most common values, and the frequencies of the most common values in the **postal\_code** column. *(Reference: PostgreSQL Community pg\_stats)*

Analyzing more details on this information:

* The **distinct\_rate** shows the proportion of distinct values versus the total rows. In the case of a unique value column, such as the Primary Key, this rate is 100%, and it is represented with the value -1. The table column from the example is near -1, meaning almost all the values are distinct.
* The **most\_common\_values** gives insight into the **postal\_code** values that repeat the most. In this case, four values: **null, 22474, 9668**, and **52137**.
* Finally, the **most\_common\_frequencies** shows the frequency of each of the most common values, the nulls are 0.66% of the total values, and each of the other values represents 0.33% of the total values from the sample.

We can verify the information the next way:

pagila=*#* WITH total AS (SELECT count(\*)::numeric cnt FROM address) SELECT

postal\_code, count(address.\*), round(count(address.\*)::numeric \*

100 / total.cnt, 2) AS “percentage” FROM address, total

GROUP BY address.postal\_code, total.cnt HAVING count(address.\*) > 1

ORDER BY 2 DESC;

postal\_code | count | percentage

*-------------+-------+------------*

|  | | | 4 | | | 0.66 |
| --- | --- | --- | --- | --- |
| 22474 | | | 2 | | | 0.33 |
| 52137 | | | 2 | | | 0.33 |
| 9668 | | | 2 | | | 0.33 |

(4 rows)

We can see only the **null** and the other three **postal\_code** values appear more than once in the whole table, appearing four times or twice, so excepting these, all the other values are distinct. Also, by doing some math (**count x 100 / total\_rows**), we get the percentage the number of appearances represents versus the total number of values, that is the frequency.

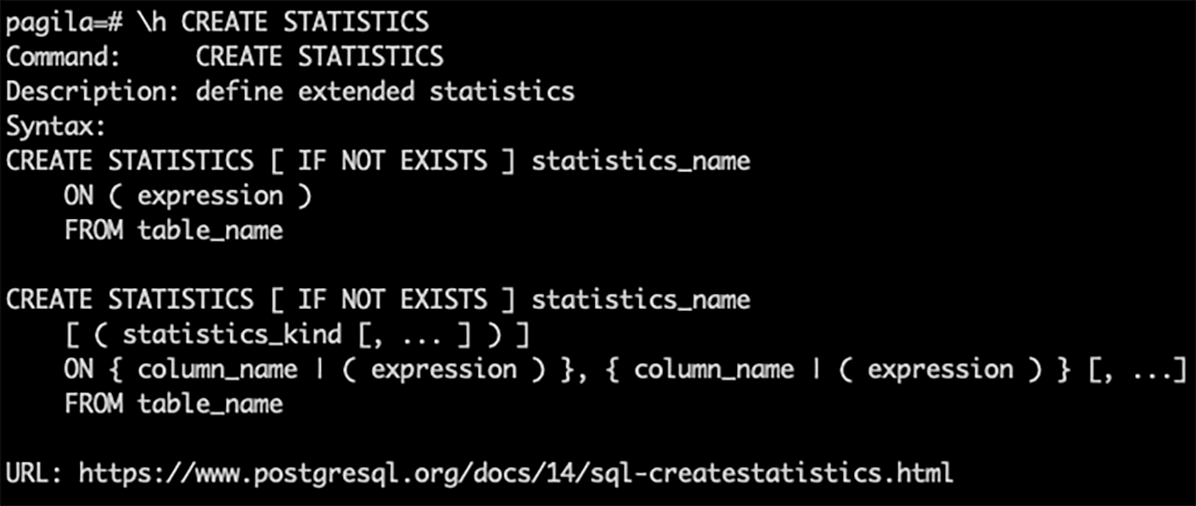
The planner uses these and the other details from the **pg\_statistics** catalog to resolve queries using the **WHERE** clause, so it can define the best path to retrieve just a fraction of the rows when required.

### Statistics in pg\_statistics\_ext\_data

So far, we have seen PostgreSQL keeps general statistics about the total number of rows per table, the space they occupy on the disk, and information about the selectivity per table column. However, there are cases where the queries correlate multiple different columns, and the single-column orientation of the regular statistics could be improved.

For such cases, PostgreSQL has the ability to compute *extended statistics*. These are a particular type of statistics and are not gathered by default, so the user or database administrator has to define them. The goal is to define the correlation between different columns, so the planner can improve its calculation when resolving queries. The way the data from different columns is related depends on the design of the database model and the tables, which is why human intervention is required.

We have to use the **CREATE STATISTICS** command to define the *extended statistics*. *Figure 13.1* shows the options for this command:



***Figure 13.1:*** *CREATE STATISTICS command options.*

There are three types of extended statistics, each covering a distinct kind of column

values correlation:

* Functional dependencies.
* Number of distinct values counts.
* Most common values list.

### Functional dependencies

This kind of correlation is possible when the knowledge about one column is sufficient to determine another. This information lets the planner make accurate estimations about the total number of rows when the two columns are involved, for example, when using AND. Considering the same sample database as before, we can try this on the **city** table.

pagila=*#* CREATE STATISTICS extsts (dependencies) ON city\_id*,* country\_ id FROM city;

pagila=*#* ANALYZE city; pagila=*#* SELECT

stxname AS “ext\_stats\_name”, stxkeys AS “ext\_stats\_columns”,

stxddependencies AS “stats\_dependencies”

FROM pg\_statistic\_ext JOIN pg\_statistic\_ext\_data ON (oid = stxoid)

WHERE stxname = ‘extsts’;

-[ RECORD 1 ]*------+-----------------------------------------*

ext\_stats\_name | extsts ext\_stats\_columns | 1 3

stats\_dependencies | {“1 => 3”: 1.000000, “3 => 1”: 0.070000}

The **stats\_dependencies** value shows that the column **city\_id** (number 1) can 100%

identify the **country\_id** (number 3), whereas the other way, only 7%.

These estimations are beneficial, but they have some limitations. The planner assumes the dependencies on the columns are compatible and redundant, so it will do wrong estimations if they are incompatible. *Figure 13.2* shows how the planner will estimate 1 row even when the correlation is wrong **(NOTE: We will study EXPLAIN command later in this chapter).**

***Figure 13.2:*** *Dependencies extended statistics, wrong estimation.*

### Number of distinct values counts

The standard statistics rely on statical data for individual columns, which could lead to wrong estimations about the number of distinct values from a multiple columns combination.

The Number of Distinct values count (or N-Distinct) extended statistics can help when the queries request distinct values from combining different columns, for example, when using the GROUP BY clause.

We can try this type of statistic on the **pagila** example database. Let’s use the **address** table. Imagine a set of queries selecting distinct values from the **district, city\_ id**, and **postal\_code** columns; we might define the n-distinct extended statistics as follows.

pagila=*#* CREATE STATISTICS extsts2 (ndistinct) ON district, city\_id, postal\_code FROM address;

pagila=*#* ANALYZE address; pagila=*#* SELECT

stxname AS “ext\_stats\_name”, stxkeys AS “ext\_stats\_columns”, stxdndistinct AS “stats\_ndistinct”

FROM pg\_statistic\_ext JOIN pg\_statistic\_ext\_data ON (oid = stxoid) WHERE stxname = ‘extsts2’;

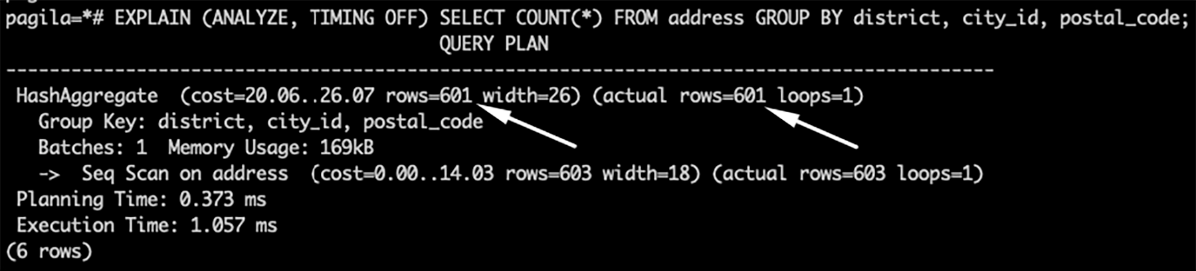
-[ RECORD 1 ]*-----+--------------------------------------------------*



ext\_stats\_name | extsts2 ext\_stats\_columns | 4 5 6

stats\_ndistinct | {“4, 5”: 601, “4, 6”: 601, “5, 6”: 601, “4, 5, 6”:

601}

Consulting the information stored in the **pg\_statistic\_ext\_data** catalog, we can verify the number of distinct values through the different **district** (number 4), **city\_id** (number 5), and **postal\_code** (number 6) column combinations. All of them produce 601 different values with the current data. *Figure 13.3* illustrates the planer does a correct estimation.

***Figure 13.3:*** *N-Distinct extended statistics.*

### Most common values list

As we saw at the beginning of this subsection, the standard statistics stored in the **pg\_ statistics** catalog contain information about the most common values per column and their frequency. This information is good when filtering data from a table based on a single column, but this can lead to lousy planning when more columns are involved.

The Most common values list (MVC List) extended statistics can improve the planner’s decisions when working with queries doing filtering on multiple column conditions. The following shows how to create this statistic on the address table.

pagila=*#* CREATE STATISTICS extsts3 (mcv) ON district, postal\_code FROM address;

pagila=*#* ANALYZE address; pagila=*#* SELECT itms.\*

FROM pg\_statistic\_ext JOIN pg\_statistic\_ext\_data ON (oid = stxoid), pg\_mcv\_list\_items(stxdmcv) itms

WHERE stxname = ‘extsts3’;

-[ RECORD 1 ]*--+------------------------------*

| index | | | 0 |
| --- | --- | --- |
| values | | | {QLD,””} |
| nulls | | | {f,f} |
| frequency | | | 0.003316749585406302 |
| base\_frequency | | | 2.200165562458575e-05 |

-[ RECORD 2 ]*--+------------------------------*

index | 1

values | {Alberta,””}

nulls | {f,f}

frequency | 0.003316749585406302 base\_frequency | 2.200165562458575e-05

-[ RECORD 3 ]*--+------------------------------*

| index | | | 2 |
| --- | --- | --- |
| values | | | {“”,65952} |
| nulls | | | {f,f} |
| frequency | | | 0.001658374792703151 |
| base\_frequency | | | 8.250620859219656e-06 |

-[ RECORD 4 ]*--+------------------------------*

index | 3

values | {“Abu Dhabi”,41136}

nulls | {f,f}

frequency | 0.001658374792703151 base\_frequency | 5.500413906146438e-06

...

Consulting the information stored for this extended statistics object, we can see the combination of **values**, their **frequency** as combined, and the **base\_frequency**, which is the result computed as per-column frequency. For example, record number 4 shows the combination of **district** and **postal\_code {“Abu Dhabi”,41136**} has a frequency of 0.16% and if the are computed per column, the frequency is only 0.0005%, which is a remarkable difference.

## Explain plan

Before understanding explain plan, let’s review what the **ANALYZE** command is. As we saw in the previous section, the **ANALYZE** command gathers statistics about the contents of tables in the database and stores this metadata in the **pg\_class**, **pg\_ statistic**, or **pg\_statistics\_ext\_data** system catalogs. Eventually, the query planner uses this metadata which helps in determining the effective execution plans for queries.

**EXPLAIN PLAN** as the name suggests gives details about the queries’ execution plan. It gives appropriate results when the statistics are updated. That is why it is recommended use **EXPLAIN PLAN** with **ANALYZE** command so that all metadata is up- to-date when the query plan is created.

Syntax as per PostgreSQL Documentation *(Reference: PostgreSQL Community Explain Plan).*

EXPLAIN [ ( option [, ...] ) ] statement EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

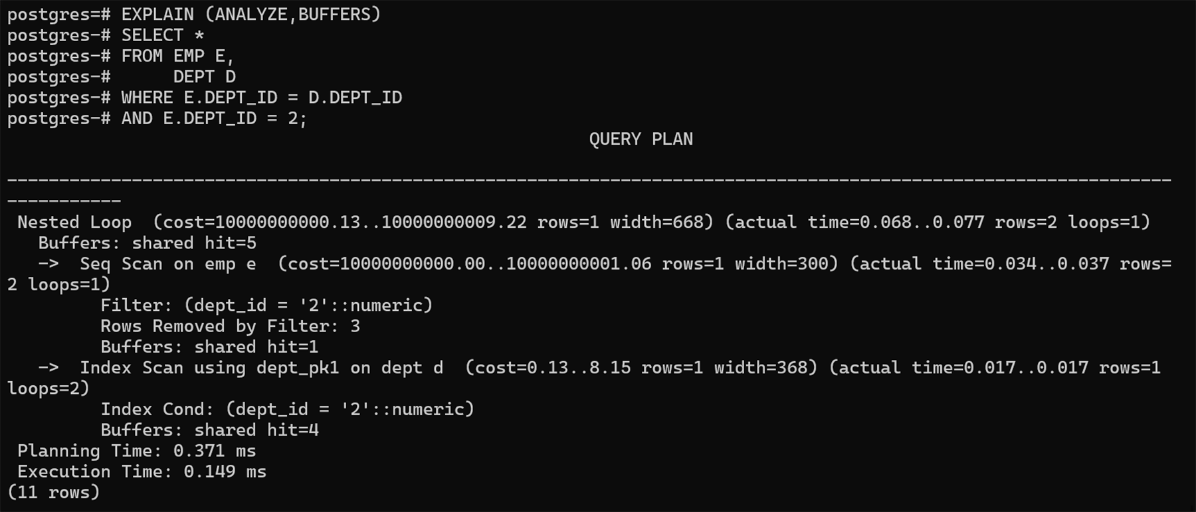
where option can be one of:

ANALYZE [ boolean ] VERBOSE [ boolean ] COSTS [ boolean ] SETTINGS [ boolean ] BUFFERS [ boolean ] WAL [ boolean ] TIMING [ boolean ] SUMMARY [ boolean ]

FORMAT { TEXT | XML | JSON | YAML }

The explain plan gives the details about whether it will use index scan or sequential scan, what plans are used to fetch the data if multiple tables are used, how many rows will come in the output of the query and many such details which can help in understanding the time taken by each step. This will help to tune to query in case there are performance issues.

Let’s see the same example of one Department (**dept**) having multiple Employees in the **emp** table we used in Chapter 11 and Chapter 12. *Figure 13.4* shows the example of the join query for the **emp** and **dept** tables to fetch details of employees in **dept\_id = 2**



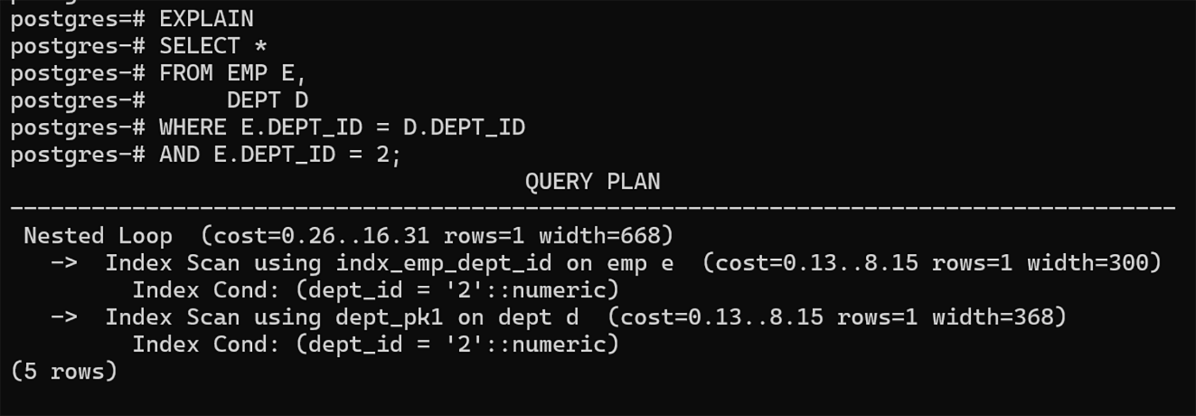
***Figure 13.4:*** *Example of EXPLAIN PLAN in a join query with ANALYZE and BUFFERS*

Notice that Sequential Scan is being used for the **WHERE** clause condition **e.dept\_id**

**=2**. There is a scope of tunning here where we can create an index on the Foreign Key column **dept\_id** of the **emp** table. Let’s create an index on this column and check the explain plan again.

CREATE INDEX INDX\_EMP\_DEPT\_ID ON EMP(DEPT\_ID);

*Figure 13.5* shows the simple Explain Plan with no attributes. Be aware of the difference in the output of the explain plan in *Figures 13.4* and *13.5.*

Notice that Index Scan is used this time with the newly created index instead of Sequential Scan. This makes the performance of the query faster. We might not be able to see the difference when there are fewer records in the query output, but the increase in performance can be observed when the data increases.

***Figure 13.5:*** *Example of EXPLAIN PLAN in a join query without any attributes*

Below gives details of the useful parameters which should be kept in mind while tunning the queries and analyzing them using the explain plan. There are more configurations from the ones mentioned below, which could be checked using the **SHOW** command or in the **postgresql.conf** file, as we will see in the next section.

* **enable\_indexscan (boolean)**
* **enable\_nestloop (boolean)**
* **enable\_seqscan (boolean)**
* **enable\_sort (boolean)**

These are self-explanatory from the name of the parameter. By default, most of such

parameters are on, and one can disable them depending upon the situation.

## Best practices for the postgresql.conf parameters

When configuring a new PostgreSQL cluster or working with an existing one to improve performance, we can consider some standard parameter configurations widely advised as best practices for almost any system. These parameters are defined in the **postgresql.conf** file, generally located in the **$PGDATA** directory.

We need to remember that these best practices or baselines are generally effective and produce the desired results; however, there is always room for improvements, especially if the database model design or user workloads have some specifics. So, we can start with the best practices and develop a regular habit of monitoring our systems and tuning what is required over time.

The following is a list of the main configuration parameters we can tune based on

best practices to bring good performance.

* **shared\_buffers**
* **work\_mem**
* **autovacuum**
* **effective\_cache\_size**
* **maintenance\_work\_mem**
* **max\_connections**

In the previous chapters, we have seen a few of the above; now, we will review them

and learn about their suggested initial values.

### shared\_buffers

This parameter defines the size of the memory area named the same, which we also might know as the “database cache.” This memory area will keep the most acceded rows so new reads can retrieve them from here rather than going to disk; also, all the data changes are written in this area instead of immediately on disk.

The default value of this parameter is very conservative, just 128 MB. The best practice for this parameter is to set it between 15% and 25% of the total RAM. So, for example, in a 64 GB RAM system, the **shared\_buffers** should be set to 16 GB. Changing this parameter demands a server restart.

### work\_mem

This parameter determines the size of the memory area named the same. Differently from the **shared\_buffers**, which exists just one for the whole database cluster (shared by all the existing databases), a **work\_mem** is allocated per user session. This area is used for all the sort operations, such as **ORDER BY, DISTINCT**, and **MERGE JOINS**.

If the operation using the **work\_mem** requires more space to complete, then PostgreSQL will use temporary files written on disk. Tuning this parameter to avoid the IO disk operations can improve performance.

The default value of this parameter is just 4 MB. We must consider the available resources and the number of concurrent user sessions to adjust it.

To be on the safe side, we can pick the **max\_sessions** value, which also should be sized right. So, the calculation for this parameter can be expressed the following way: **work\_mem = 25% of total RAM / max\_connextions.**

We can change this parameter without a server restart and even set it at the role (user) level using the **ALTER USER** command. *(Reference: PostgreSQL Community alter user).*

### autovacuum

This parameter controls if the autovacuum background process is enabled or not. As we have seen in previous chapters, vacuuming the tables and indexes is vital in a PostgreSQL system. By default, this parameter is enabled, so the recommendation is to keep it this way.

If, for any reason, you need to disable the autovacuum, the recommendation is to disable it by table rather than for the whole cluster, changing it at the cluster level

in the postgresql.conf file requires a server restart. You can use the **ALTER TABLE** command to disable the autovacuum at the table level. *(Reference: PostgreSQL Community alter table).*

### effective\_cache\_size

This parameter lets the query planner know how much memory is expected to be available in the system for disk caching within the database. It does not represent an allocated memory area, but the planner uses its value to decide whether the operations to resolve specific queries will fit the RAM.

If the value is too low, the planner might disregard the index scans and prefer sequential table scans, which usually perform slower to access specific data subsets.

The default for this parameter is 4 GB, and the best practice is setting it between 50% and 75% of the total RAM. Change it doesn’t require a server restart.

### maintenance\_work\_mem

This parameter defines the size for the memory area named the same, which is used for the maintenance tasks **VACUUM, CREATE INDEX**, and **ALTER TABLE ADD FOREIGN KEY**. Since only one of these tasks can be running simultaneously per connection and are usually executed in a lower proportion than the **work\_mem** operations, its value can be significantly larger than **work\_mem.**

Its default value is 64 MB, which is very conservative. The advisable value for this

parameter is between 5% and 10% of the total RAM.

### max\_connections

This parameter limits the maximum number of simultaneous user database connections. As we saw before, each of these connections can hold a **work\_mem** size memory area at least.

Remember, PostgreSQL has a process-based architecture, so each user session represents a new process at the database server. There is some relation between the number of CPUs available in the system and the maximum number of processes it can handle efficiently.

The default of this parameter is 100, and usually, it is recommended to set this value lower than 10 per CPU and consider 20 per CPU as the limit. If you need to allow more user sessions than these, consider a pooler such as **pgbouncer**.

## Summary

The above parameters are the main ones we advise you to verify when working to improve the performance of your system. Next is a summary table with the default and suggested values.

| **Parameter** | **Default values** | **Best practice value** |
| --- | --- | --- |
| **shared\_buffers** | 128 MB | Between 15 - 25% of total RAM |
| **work\_mem** | 4 MB | 25% of total RAM / max\_connextions |
| **autovacuum** | ON | ON |
| **effective\_cache\_size** | 4 GB | Between 50% - 75% of total RAM |
| **maintenance\_work\_mem** | 64 MB | Between 5% - 10% of total RAM |
| **max\_connections** | 100 | Up to 20 per CPU |

Upgrade your database:

**To check if your system is compatible?**

/usr/pgsql-11/bin/pg\_upgrade -d /u01/pgsql/10 -D /u01/pgsql/11 -b /usr/local/pgsql/bin/ -B /usr/pgsql-11/bin/ -c

**To upgrade using copy method?**

/usr/pgsql-11/bin/pg\_upgrade -d /u01/pgsql/10 -D /u01/pgsql/11 -b /usr/local/pgsql/bin/ -B /usr/pgsql-11/bin/

**To upgrade using link method?**

/usr/pgsql-11/bin/pg\_upgrade -d /u01/pgsql/10 -D /u01/pgsql/11 -b /usr/local/pgsql/bin/ -B /usr/pgsql-11/bin/ -k

pgbadger

<https://severalnines.com/database-blog/postgresql-log-analysis-pgbadger>

grafana

<https://postgreshelp.com/configure-grafana-for-postgresql/>