Introduction

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

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

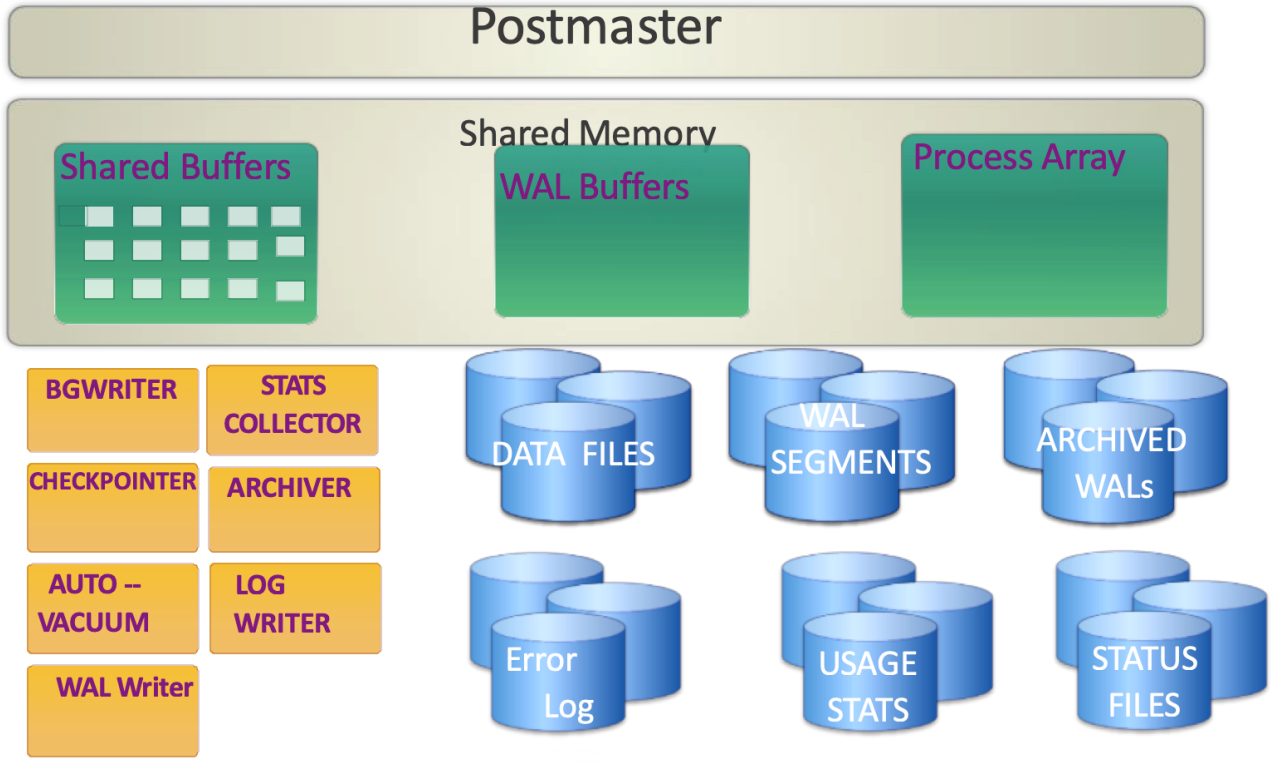
Database Limits

Item	Upper Limit	Comment
database size	unlimited	
number of databases	4294950911	
relations per database	1431650303	
relation size	32 TB	with the default BLCKSZ of 8192 bytes
rows per table	limited by the number of tuples that can fit onto 4,294,967,295 pages	
columns per table	1600	further limited by tuple size fitting on a single page.
columns in a result set	1664	

field size	1 GB	
identifier length	63 bytes	can be increased by recompiling PostgreSQL
indexes per table	unlimited	constrained by maximum relations per database
columns per index	32	can be increased by recompiling PostgreSQL
partition keys	32	can be increased by recompiling PostgreSQL

Architecture

Overview of PostgreSQL Architecture



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

Postmaster

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

Utility Processes

Background writer
Writes updated data blocks to disk
WAL writer
Flushes write-ahead log to disk
Checkpointer process
Automatically performs a checkpoint based on config parameters
Autovacuum launcher
Starts Autovacuum workers as needed
Autovacuum workers
Recover free space for reuse
Logging collector
Routes log messages to syslog, eventlog, or log files
Archiver
Archives write-ahead log files

Installation Directory Layout

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

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

Data Directory Layout

Item	Description
PG_VERSION	A file containing the major version number of PostgreSQL

base	Subdirectory containing per-database subdirectories (pg_default tablespace)
current_logfiles	File recording the log file(s) currently written to by the logging collector
global	Subdirectory containing cluster-wide tables, such as pg_database
pg_commit_ts	Subdirectory containing transaction commit timestamp data
pg_dynshmem	Subdirectory containing files used by the dynamic shared memory subsystem
pg_logical	Subdirectory containing status data for logical decoding
pg_multixact	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify	Subdirectory containing LISTEN/NOTIFY status data
pg_replslot	Subdirectory containing replication slot data
pg_serial	Subdirectory containing information about committed serializable transactions
pg_snapshots	Subdirectory containing exported snapshots
pg_stat	Subdirectory containing permanent files for the statistics subsystem
pg_stat_tmp	Subdirectory containing temporary files for the statistics subsystem
pg_subtrans	Subdirectory containing subtransaction status data
pg_tblspc	Subdirectory containing symbolic links to tablespaces
pg_twophase	Subdirectory containing state files for prepared transactions
pg_wal	Subdirectory containing WAL (Write Ahead Log) files
pg_xact	Subdirectory containing transaction commit status data

pg_hba.conf	File used to store Server access ACLs
pg_ident.conf	User name maps are defined in this file and used by Authentication methods such as IDENT, GSSAPI
postgresql.conf	A file used for storing configuration parameters
postgresql.auto.conf	A file used for storing configuration parameters that are set by ALTER SYSTEM
postmaster.opts	A file recording the command-line options the server was last started with
postmaster.pid	A lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (empty on Windows), first valid listen_address (IP address or *, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown)

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

Write a head logging

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

Rollback and Undo

- Dynamic allocation of disk space is used for the storage and processing of records within tables.

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

Data File Storage Internals

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

Free Space and Visibility Map Files

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

Background Writer

All data pages read from data files are saved in the shared buffer first and then they are modified in shared buffer. Usually, the modified pages remain in shared buffer and are reused by server processes when they need same data however at some point the shared buffer will fill up then postgres starts evicting the pages using LRU mechanism during CHECKPOINT time or by Server processes this may cause some processes to wait until they get free pages in shared buffer.

In version 8.0 a new process called as Background Process (BGWriter) is introduced to address this issue. BGWriter sleeps until **bgwriter_delay** time and wakes to make a round around the shared buffer to select and flush some dirty pages specified by **bgwriter_lru_maxpages**. With this mechanism, BGWriter ensures there are always clean pages available for upcoming transactions.

Settings for BGWriter

bgwriter_delay : Time delay between each round

bgwriter_lru_maxpages : Maximum pages to be flushed in each round.

bgwriter_flush_after : After this many bytes request OS to flush data to disk.

Checkpointing [CHECKPOINT]

Checkpoint is the time within database time that all the changes done are written to data files. There are several advantages with this approach.

- All changes are written to data files during checkpoint time instead of as they are done. This will reduce the disk IO and make transactions complete faster.
- IT will mark a Time within WAL time line that the data is consistent to this point and in even of a crash PostgreSQL knows it has to perform recovery from latest checkpoint.

When check points are triggered:

- Check point occurs every **checkpoint_timeout** interval.
- When the size of WALs reaches **max_wal_size** (default 1GB)
- When **pg_basebackup** command is executed
- When **pg_start_backup()** is executed for online filesystem backup
- When DBA executes the **checkpoint** command manually

settings for checkpointing

checkpoint_timeout : Specifies checkpoint interval (default is 300s or 5m)

checkpoint_completions_target : specifies a fraction of checkpoint time interval as checkpoint completions time (default 0.9 i,e 90% of default 5 minutes).

checkpoint_flush_after : After writing the much data forces OS to issue writes to underlying storage like disk, RAID etc.

Checkpoint are designed to write all dirty pages to disk but on some busy systems this may cause significant IO load that's the reason checkpoint are throttled to start at checkpoint start time and end before next checkpoint.

WAL (Write ahead log)

WALs or Write Ahead Log is a necessary component of MVCC and ACID compliance as the transactions are committed on the server they are written first to WAL Segments (physical WAL file on disk) residing in pg_wal directory and afterwards they are written to respective data files during checkpointing or by BGWriter process. This ensures that the writes to disk happens in a controlled manner rather a flurry of writes to disk as transaction are committed.

- WAL guarantees durability of transaction
- They help reduce transaction latency
- They help in Database Cluster recovery
- They play central role in all types replications supported by PostgreSQL

settings for WAL

wal_level : Sets the WAL level, there are 3 levels of WAL namely **'minimal', 'replica'** and **'logical'.** This setting determines the volume of WAL generated per transaction. 'minimal being less and logical being more.

wal_sync_method : PostgreSQL allows 4 different WAL sync methods namely **'open_datasync', 'fdatasync', 'sync', 'open_sync'** and '**fsync_writethrough**' .

Default on Linux and FreeBSD Unix is '**fdatasync**'. Default may not be ideal we can do a **pg_test_fsync** command line tools to run a sync test using each of the sync method and report the timings. DBA can select the latest sync method by comparing the data returned by the **pg_test_fsync** command

max_wal_size : This setting determines how much WAL file needs to generated before the old entries are discarded default is 1GB. However is this is not hard limit, PostgreSQL allow the WAL growth beyond this size limit incase of heavy load, replication slots or a failing archive_command.

A lesser **max_wal_size** may trigger checkpoint before it is due time with a warning written to error log file. DBA needs to observe the error log and may needs to increase the **max_wal_size.**

min_wal_size : During the checkpoint old WAL files are removed rather reused. This setting allows retention of minimum this much size WAL segments to take any spikes in the transactions. default is 80MB.

Advance Monitoring

Monitoring check Points

pg_stat_checkpointer

The pg_stat_checkpointer view will always have a single row, containing data about the checkpointer process of the cluster.

Column Type Description
num_timed bigint Number of scheduled checkpoints due to timeout
num_requested bigint Number of requested checkpoints
num_done bigint Number of checkpoints that have been performed
restartpoints_timed bigint Number of scheduled restartpoints due to timeout or after a failed attempt to perform it
restartpoints_req bigint Number of requested restartpoints
restartpoints_done bigint Number of restartpoints that have been performed
write_time double precision Total amount of time that has been spent in the portion of processing checkpoints and restartpoints where files are written to disk, in milliseconds
sync_time double precision Total amount of time that has been spent in the portion of processing checkpoints and restartpoints where files are synchronized to disk, in milliseconds
buffers_written bigint Number of shared buffers written during checkpoints and restartpoints
slru_written bigint Number of SLRU (Simple Least Recently Used)buffers written during checkpoints and restartpoints
stats_reset timestamp with time zone Time at which these statistics were last reset

When to Increase the WAL_Buffers or max_wal_size

- When too many requested checkpoints are happening as it indicate either WAL_Buffer or MAX_WAL_SIZE is inadequate
- Error Log file logs all checkpoints by default and also reports what causing them

- If checkpoints are Forced (user running CHECKPOINT command)
- If WAL_Buffers are full
- MAX_WAL_SIZE is reached which forced the check point

pg_stat_bgwriter

The `pg_stat_bgwriter` view will always have a single row, containing data about the background writer of the cluster.

Column Type Description
buffers_clean bigint Number of buffers written by the background writer
maxwritten_clean bigint Number of times the background writer stopped a cleaning scan because it had written too many buffers
buffers_alloc bigint Number of buffers allocated
stats_reset timestamp with time zone Time at which these statistics were last reset

pg_stat_database

The pg_stat_database view will contain one row for each database in the cluster, plus one for shared objects, showing database-wide statistics.

Column Type Description
datid oid OID of this database, or 0 for objects belonging to a shared relation
datname name Name of this database, or NULL for shared objects.
numbackends integer Number of backends currently connected to this database, or NULL for shared objects. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
xact_commit bigint Number of transactions in this database that have been committed

Column Type Description
xact_rollback bigint Number of transactions in this database that have been rolled back
blks_read bigint Number of disk blocks read in this database
blks_hit bigint Number of times disk blocks were found already in the buffer cache, so that a read was not necessary (this only includes hits in the PostgreSQL buffer cache, not the operating system's file system cache)
tup_returned bigint Number of live rows fetched by sequential scans and index entries returned by index scans in this database
tup_fetched bigint Number of live rows fetched by index scans in this database
tup_inserted bigint Number of rows inserted by queries in this database
tup_updated bigint Number of rows updated by queries in this database
tup_deleted bigint Number of rows deleted by queries in this database
conflicts bigint Number of queries canceled due to conflicts with recovery in this database. (Conflicts occur only on standby servers; see pg_stat_database_conflicts for details.)
temp_files bigint Number of temporary files created by queries in this database. All temporary files are counted, regardless of why the temporary file was created (e.g., sorting or hashing), and regardless of the **log_temp_files** setting.
temp_bytes bigint Total amount of data written to temporary files by queries in this database. All temporary files are counted, regardless of why the temporary file was created, and regardless of the **log_temp_files** setting.
deadlocks bigint Number of deadlocks detected in this database
checksum_failures bigint Number of data page checksum failures detected in this database (or on a shared object), or NULL if data checksums are disabled.

Column Type Description
checksum_last_failure timestamp with time zone Time at which the last data page checksum failure was detected in this database (or on a shared object), or NULL if data checksums are disabled.
blk_read_time double precision Time spent reading data file blocks by backends in this database, in milliseconds (if **track_io_timing** is enabled, otherwise zero)
blk_write_time double precision Time spent writing data file blocks by backends in this database, in milliseconds (if **track_io_timing** is enabled, otherwise zero)
session_time double precision Time spent by database sessions in this database, in milliseconds (note that statistics are only updated when the state of a session changes, so if sessions have been idle for a long time, this idle time won't be included)
active_time double precision Time spent executing SQL statements in this database, in milliseconds (this corresponds to the states active and fastpath function call in pg_stat_activity)
idle_in_transaction_time double precision Time spent idling while in a transaction in this database, in milliseconds (this corresponds to the states idle in transaction and idle in transaction (aborted) in pg_stat_activity)
sessions bigint Total number of sessions established to this database
sessions_abandoned bigint Number of database sessions to this database that were terminated because connection to the client was lost
sessions_fatal bigint Number of database sessions to this database that were terminated by fatal errors
sessions_killed bigint Number of database sessions to this database that were terminated by operator intervention
parallel_workers_to_launch bigint Number of parallel workers planned to be launched by queries on this database
parallel_workers_launched bigint Number of parallel workers launched by queries on this database
stats_reset timestamp with time zone Time at which these statistics were last reset

pg_stat_database_conflicts

The pg_stat_database_conflicts view will contain one row per database, showing database-wide statistics about query cancels occurring due to conflicts with recovery on standby servers. This view will only contain information on standby servers, since conflicts do not occur on primary servers.

Column Type Description
datid oid OID of a database
datname name Name of this database
confl_tablespace bigint Number of queries in this database that have been canceled due to dropped tablespaces
confl_lock bigint Number of queries in this database that have been canceled due to lock timeouts
confl_snapshot bigint Number of queries in this database that have been canceled due to old snapshots (depcrecated)
confl_bufferpin bigint Number of queries in this database that have been canceled due to pinned buffers
confl_deadlock bigint Number of queries in this database that have been canceled due to deadlocks
confl_active_logicalslot bigint Number of uses of logical slots in this database that have been canceled due to old snapshots or too low a **wal_level** on the primary

pg_stat_wal

The pg_stat_wal view will always have a single row, containing data about WAL activity of the cluster.

Column Type Description
wal_records bigint Total number of WAL records generated
wal_fpi bigint Total number of WAL full page images generated

Column Type Description
wal_bytes numeric Total amount of WAL generated in bytes
wal_buffers_full bigint Number of times WAL data was written to disk because WAL buffers became full
stats_reset timestamp with time zone Time at which these statistics were last reset

pg_stat_statement

This EXTENTION provides a statement statistics for all the statements running on the server.

This module's libraries needs to be loaded before can be used. In **postgresql.conf** file set **shared_preload_libraries = pg_stat_statements** and server restart is needed.

Then in the database (eg postgres) from where you want to monitor create extension.

```
CREATE EXTENTION pg_stat_statements;
```

Which adds the extensions and creates a view called as **pg_stat_statement** and saves statement wise statistics in it.

pg_stat_statement view contains following columns.

Column Type Description
userid oid (references pg_authid.oid) OID of user who executed the statement
dbid oid (references pg_database.oid) OID of database in which the statement was executed
queryid bigint Internal hash code, computed from the statement's parse tree
query text

Text of a representative statement
plans bigint Number of times the statement was planned (if `pg_stat_statements.track_planning` is enabled, otherwise zero)
total_plan_time double precision Total time spent planning the statement, in milliseconds (if `pg_stat_statements.track_planning` is enabled, otherwise zero)
min_plan_time double precision Minimum time spent planning the statement, in milliseconds (if `pg_stat_statements.track_planning` is enabled, otherwise zero)
max_plan_time double precision Maximum time spent planning the statement, in milliseconds (if `pg_stat_statements.track_planning` is enabled, otherwise zero)
mean_plan_time double precision Mean time spent planning the statement, in milliseconds (if `pg_stat_statements.track_planning` is enabled, otherwise zero)
stddev_plan_time double precision Population standard deviation of time spent planning the statement, in milliseconds (if `pg_stat_statements.track_planning` is enabled, otherwise zero)
calls bigint Number of times the statement was executed
total_exec_time double precision Total time spent executing the statement, in milliseconds
min_exec_time double precision Minimum time spent executing the statement, in milliseconds
max_exec_time double precision

Maximum time spent executing the statement, in milliseconds
mean_exec_time double precision Mean time spent executing the statement, in milliseconds
stddev_exec_time double precision Population standard deviation of time spent executing the statement, in milliseconds
rows bigint Total number of rows retrieved or affected by the statement
shared_blks_hit bigint Total number of shared block cache hits by the statement
shared_blks_read bigint Total number of shared blocks read by the statement
shared_blks_dirtied bigint Total number of shared blocks dirtied by the statement
shared_blks_written bigint Total number of shared blocks written by the statement
local_blks_hit bigint Total number of local block cache hits by the statement
local_blks_read bigint Total number of local blocks read by the statement
local_blks_dirtied bigint Total number of local blocks dirtied by the statement
local_blks_written bigint Total number of local blocks written by the statement

temp_blks_read bigint Total number of temp blocks read by the statement
temp_blks_written bigint Total number of temp blocks written by the statement
blk_read_time double precision Total time the statement spent reading blocks, in milliseconds (if **track_io_timing** is enabled, otherwise zero)
blk_write_time double precision Total time the statement spent writing blocks, in milliseconds (if **track_io_timing** is enabled, otherwise zero)
wal_records bigint Total number of WAL records generated by the statement
wal_fpi bigint Total number of WAL full page images generated by the statement
wal_bytes numeric Total amount of WAL generated by the statement in bytes

This extension also provides statistics reset function to for managing statistics

`pg_stat_statements_reset(userid Oid, dbid Oid, queryid bigint) returns void`

`pg_stat_statements_reset` discards statistics gathered so far by pg_stat_statements extension corresponding to the specified userid, dbid and queryid combination.

pg_buffercache

This extension also provided by the CONTRIB module. The pg_buffercache extension tracks usage of each buffer page in shared pool and aves that data in pg_buffercache view which is also created by the extension.

The pg_buffercache view contains one row for each buffer page present in shared_buffer.

Run the following command from postgres prompt to create this extension.

```
postgres=# CREATE EXTENSION pg_buffercache;
```

Needs to be created in all databases where tableware shared_buffer usage monitoring is needed.

pg_buffercache view details

Column Type Description
bufferid integer ID, in the range 1..`shared_buffers`
`relfilenode` `oid` (references `pg_class`.`relfilenode`) Filenode number of the relation
reltablespace oid (references pg_tablespace.oid) Tablespace OID of the relation
`reldatabase` `oid` (references `pg_database`.`oid`) Database OID of the relation
relforknumber smallint Fork number within the relation; see `include/common/relpath.h`
relblocknumber bigint Page number within the relation
isdirty boolean Is the page dirty?
usagecount smallint Clock-sweep access count

pinning_backends integer Number of backends pinning this buffer

Query to get details of shared_buffer usage per **schema.table**

```
regression=# SELECT n.nspname, c.relname, count(*) AS buffers
             FROM pg_buffercache b JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             JOIN pg_namespace n ON n.oid = c.relnamespace
             GROUP BY n.nspname, c.relname
             ORDER BY 3 DESC
             LIMIT 10;
nspname   |        relname         | buffers
-----------+------------------------+---------
 public     | delete_test_table      |     593
 public     | delete_test_table_pkey |     494
 pg_catalog | pg_attribute           |     472
 public     | quad_poly_tbl          |     353
 public     | tenk2                  |     349
 public     | tenk1                  |     349
 public     | gin_test_idx           |     306
 pg_catalog | pg_largeobject         |     206
 public     | gin_test_tbl           |     188
 public     | spgist_text_tbl        |     182
(10 rows)
```

pg_top

This external tool provides a top like command for the queries and processes running on postgresql server. Helps in identifying queries/process consuming maximum resources.

This is not included in standard PostgreSQL installations and needs to downloaded and installed separately.

Installing pg_top

```
# wget https://download-ib01.fedoraproject.org/pub/epel/7/x86_64/Packages/p/pg_top-3.7.0-5.el7.x86_64.rpm


# rpm -i pg_top-3.7.0-5.el7.x86_64.rpm
```

Using pg_top

invoke pg_top command from OS postgres user. It provides linus top command like output refreshing every few seconds. You can exit this screen by pressing q key from keyboard.

```
-bash-4.2$ pg_top
```

Revisit Indexes

Indexes in PostgreSQL

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

Types of Indexes Supported by PostgreSQL

Hash Indexes

B-Tree Indexes

GIN Indexes

GiST Indexes

BRIN Indexes

B-Tree Indexes

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

Operations supported by B-tree

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

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

Syntax:

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

Indexes and ORDER BY

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

Syntax:

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

This will save sorting time spent by the query

Multicolumn Index

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

This speeds up the query execution.

Syntax:

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

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

Unique Indexes

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

Syntax:

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

Currently only B-tree indexes can be declared unique.

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

Functional Indexes

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

Syntax:

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

They can also be created on user defined functions

Partial Index

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

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

Syntax:

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

Hash Indexes

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

Syntax:

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

GiST Indexes

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

Syntax:

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

GIN Indexes

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

Syntax:

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

BRIN Indexes

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

BRIN indexes are less expensive than B-Tree indexes.

Syntax:

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

Index-Only Scans and Covering Indexes

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

```
create index testidx on emp (ename) include (job, sal);
```

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

CREATE INDEX CONCURRENTLY

To create an index concurrently in PostgreSQL, allowing reads and writes to continue on the table during the index creation process.

```
CREATE INDEX CONCURRENTLY index_name ON table_name (column_name);
```

Paralle INDEX Creation

PostgreSQL 17 supports parallel index creation, which can significantly speed up the process for large indexes. Several conditions and configurations influence whether parallel index creation occurs and how many parallel workers are used:

- `max_parallel_maintenance_workers` Parameter:
    - This is the primary setting controlling the maximum number of parallel workers that can be used for maintenance operations, including index creation.
    - A value greater than 0 is required for parallel index creation to be considered.
    - The default value is typically 2. Increasing this value can allow more workers to be used, potentially reducing index creation time, provided sufficient CPU and I/O capacity.
- **Index Size and Cost Model**:
    - PostgreSQL's cost model automatically determines if parallel execution is beneficial for a given index creation operation.
    - Parallel index builds are generally considered for sufficiently large indexes where the overhead of parallelization is outweighed by the potential performance gains.
    - The cost model considers factors like the size of the table, the complexity of the index, and system resources.
- `maintenance_work_mem` Parameter:
    - This parameter defines the maximum amount of memory used by maintenance operations, including index creation.
    - Parallel index builds can benefit from increased `maintenance_work_mem`, as parallel workers require a share of this memory (at least 32MB per worker, plus 32MB for the leader process).
    - Insufficient `maintenance_work_mem` might limit the number of parallel workers that can be requested.
- `parallel_workers` Storage Parameter (via ALTER TABLE):
    - You can explicitly set the `parallel_workers` storage parameter for a table using ALTER TABLE.

    - This directly controls the number of parallel worker processes requested for index builds on that specific table, bypassing the cost model.
    - Setting `parallel_workers` to 0 for a table will disable parallel index builds for that table.
- **CPU Capacity and I/O Performance**:
    - Parallel index creation relies on available CPU resources and efficient I/O.
    - If the system is already I/O bound or lacks sufficient CPU capacity, increasing `max_parallel_maintenance_workers` or `parallel_workers` may not yield significant performance improvements.

Parallel Index Creation vs Concurrent Index Creation:

Summary of Differences

Feature	Purpose	Locking Behavior	Performance	Command Syntax
Parallel	Speed up the index build using multiple cores.	Acquires a ShareLock (allows other non-concurrent index builds and reads, but blocks writes with default CREATE INDEX).	Faster build time on suitable hardware.	Implicitly used by CREATE INDEX, configurable via parameters.
Concurrent	Avoid blocking INSERT/UPDATE/DELETE operations during the build.	Acquires minimal locks (allows reads/writes, blocks other concurrent index builds).	Slower build time due to extra overhead.	CREATE INDEX CONCURRENTLY.

HypoPG Extension

HypoPG is an open-source PostgreSQL extension that enables the creation of **hypothetical (or virtual) indexes**. These are simulated indexes that exist only in memory for the duration of a session or connection—they consume no disk space, CPU, or I/O resources during creation or maintenance. The PostgreSQL query planner can "see" and evaluate them during EXPLAIN (without ANALYZE), allowing you to test potential performance improvements without committing to real index builds.

- **Key Features**:
    - Supports standard CREATE INDEX syntax for hypothetical indexes.

    - Indexes are session-private (per backend connection) and automatically cleaned up on disconnect.
    - Compatible with PostgreSQL 9.2 and later (up to at least 17.x as of 2025).
    - Provides views and functions for listing, dropping, hiding, and resetting hypothetical indexes.
    - GUCs (configuration parameters): hypopg.enabled (default: on, to toggle globally), hypopg.standalone (default: off, for using real OIDs on standbys).
- **Limitations**:
    - Only works with EXPLAIN (not EXPLAIN ANALYZE, as no real execution occurs).
    - Limited to ~2,500 hypothetical indexes per session (due to OID borrowing from reserved ranges).
    - Not suitable for production testing of complex joins/aggregations—use a staging environment for that.
    - No upgrade scripts provided; reinstall if needed.

Use Cases

HypoPG shines in performance tuning and development scenarios where you want to iteratively test index ideas without overhead. Common use cases include:

Use Case	Description	Why HypoPG?
Query Optimization	Identify if an index on specific columns (e.g., WHERE clauses, JOINs) would speed up slow queries.	Quickly prototype without building real indexes, which can take hours on large tables.
Index Strategy Planning	Evaluate multiple index candidates for a problematic query before production deployment.	Saves resources; test on replicas or dev environments without disk bloat.
Hiding Existing Indexes	Simulate index drops to see if removing an underused index affects query plans.	Helps decide on index cleanup without risking regressions.
Standby Server Testing	Test indexes on read replicas without write access or resource costs.	hypopg.standalone = on allows real OIDs on standbys.
Bulk Load/ETL Prep	Pre-analyze indexes for large data imports or migrations.	Avoids locking tables during real builds; ideal for high-traffic systems.

HypoPG Installation on RHEL or Compatibles

Download RPM

```
[admin@pgserver ~]$ wget https://download.postgresql.org/pub/repos/yum/17/redhat/rhel-9-x86_64/hypopg_17-llvmjit-1.4.1-2PGDG.rhel9.x86_64.rpm
```

Install RPM

```
[admin@pgserver ~]$ sudo dnf -y localinstall hypopg_17-llvmjit-1.4.1-2PGDG.rhel9.x86_64.rpm
```

Create Extension:

```
CREATE EXTENSION hypopg;
```

```
\dx hypopg
```

Example:

Create test table and populate it with data:

```
CREATE TABLE hypo AS SELECT id, 'line ' || id AS val FROM generate_series(1, 1000000) id;
```

```
ANALYZE hypo;
```

Testing the effect of HypoPG

```
EXPLAIN SELECT val FROM hypo WHERE id = 500000;
```

```
SELECT * FROM hypopg_create_index('CREATE INDEX ON hypo (id)');
```

List hypothetical indexes

```
SELECT * FROM hypopg_list_indexes();
```

Explain a query: Planner now "sees" and uses the index

```
EXPLAIN SELECT val FROM hypo WHERE id = 500000;
```

Example (Dropping a hypothetical Index or Reset HypoPG:

Drop by OID (from hypopg_list_indexes())

```
SELECT hypopg_drop_index(12345);
```
-- Replace with actual OID

Or reset all

```
SELECT hypopg_reset();
```

Example (Hide an Existing Real Index to Test Alternatives):

Create Real Index:

```
CREATE INDEX hypo_id_val_idx ON hypo (id, val);
ANALYZE hypo;
```

Hide Real Index:

```
SELECT hypopg_hide_index('hypo_id_val_idx'::regclass);
```

Create hypo alternative

```
SELECT * FROM hypopg_create_index('CREATE INDEX ON hypo (id)');
```

Explain: Now uses the hypo index instead

```
EXPLAIN SELECT val FROM hypo WHERE id = 500000;
```

PostgreSQL Locks

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

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

Table Level Locking.

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

Explicit Table Level Locks

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

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

ALL locks are released when the transaction holding them ends.

Syntax:

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

where lockmode is one of:

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

Parameters:

name

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

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

lockmode

Specifies which lock mode to use if not specified then default is **ACCESS EXCLUSIVE**, the most restrictive mode is used.

NOWAIT

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

Table-level Lock Modes

Lets study all the table level lock modes supported by postgres.

ACCESS SHARE

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

ROW SHARE

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

ROW EXCLUSIVE

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

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

SHARE UPDATE EXCLUSIVE

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

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

SHARE

This mode protects a table against concurrent data changes.

Acquired implicitly by CREATE INDEX (without CONCURRENTLY).

SHARE ROW EXCLUSIVE

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

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

EXCLUSIVE

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

Acquired implicitly by REFRESH MATERIALIZED VIEW CONCURRENTLY.

ACCESS EXCLUSIVE

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

Acquired implicitly by the DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL, and REFRESH MATERIALIZED VIEW (without CONCURRENTLY) commands. Many forms of ALTER TABLE also acquire a lock at this level.

This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Table Level Lock compatibility chart

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

ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

an X indicates conflict and blank indicates compatible.

Row Level Locks

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

Row Level locks are released when a transactions completes.

There are 4 ROW level locks provided.

FOR SHARE
FOR UPDATE
FOR KEY SHARE
FOR NO KEY UPDATE

FOR UPDATE

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

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

FOR NO KEY UPDATE

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

FOR SHARE

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

FOR KEY SHARE

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

Lock Mode Compatibility Chart below

Requested Lock Mode	**Current Lock Mode**			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

Explicitly Acquiring the Row Level Locks

Syntax:

```
start transaction;
........
Select  .......   <Lock mode>;
.......
commit / rollback;
```

Example:

```
start transaction;
...........
select * from where deptno = 20 for update;
```

```
.....
commit / rollback;
```

Lock Monitoring

p**g_locks** system view holds the details of locks currently held or waiting by users.

Its provides global view on all locks not just for current database but for all database. Contains information about the transactions which are holding (granted) locks and also the session which are waiting for the locks.

- Shows all locks on a particular table.
- Shows all locks held by a particular session.
- Useful in determining lock contention and overall database performance.
- This view must be joined with **pg_stat_activity** view to get readable details.

Parallel Query Execution

- By default query use one process to execute
- Parallel queries uses more than one process to improve performance
- Planner does not use parallel plan for write queries or any query which locks data
- In Isolation level serializable planner ignore parallelism
- Number of processes limited by **`max_parallel_workers_per_gather`**
- **`max_parallel_workers_per_gather`** is limited by **`max_worker_processes`** and **`max_parallel_workers.`**
- Also the availability of background workers.
- In such case **`max_worker_processes`** and **`max_parallel_workers`** cab be increased

Benchmarking PostgreSQL

Pgbench tool is provided to benchmark the performance of PostgreSQL Database. By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction however you can also your own scripts files to benchmark.

Sample output from pgbench

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
maximum number of tries: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
number of failed transactions: 0 (0.000%)
latency average = 11.013 ms
latency stddev = 7.351 ms
initial connection time = 45.758 ms
tps = 896.967014 (without initial connection time)
```

Setup a schema for TCP-B test

```
pgbench -h localhost -U postgres -p 5432 -i dbname
```

(Create 4 table in the given database)

```
Table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

Run the benchmark test

```
pgbench -h localhost -U postgres -c  <client>  -t <transaction per client>  -T
<duration in seconds>   <database name>
```

Run Benchmark test with a script file.

```
pgbench -h localhost -U postgres  -f <script filename>  -c  < no of clients>  -t
<transaction per client> -T <duration in seconds>   <database name>
```