PERCONA

PostgreSQL Performance Tuning

Strategies, best practices, and tips from Percona PostgreSQL experts

Contents

PostgreSQL Performance Tuning

Strategies, best practices, and tips from Percona PostgreSQL experts

Percona, a leader in open source database solutions, has decades of expertise in managing, optimizing, and scaling databases. This eBook, PostgreSQL Performance Tuning, is a compilation of some of that knowledge. A curated selection of "greatest hits" rather than an exhaustive manual on PostgreSQL performance, it's a collection of tried-and-tested solutions that address the most pressing performance challenges.

Each chapter focuses on a distinct facet of PostgreSQL optimization, sourced from our experts respected for their deep understanding of PostgreSQL (among other leading database technologies). From the intricacies of schema design and partitioning to the benefits of query optimization, this eBook is designed with a clear objective: to serve as a companion for DBAs, developers, and system architects who aspire to improve the performance of their PostgreSQL databases.

As a testament to Percona's commitment to empowering the community by sharing knowledge and fostering innovation in open source databases, this eBook provides the PostgreSQL guidance you need whether you are looking to troubleshoot a specific issue or seeking to build a scalable database infrastructure from the ground up.

PostgreSQL Performance Tuning is your expert guide to achieving exceptional database performance, offering you the agility and knowledge to tackle your performance optimization challenges.

Parameters

Our first chapter looks at optimizing parameters in PostgreSQL, highlighting how even minor adjustments can lead to substantial improvements. Sourced from the seasoned expertise of Percona's PostgreSQL professionals, this advice will be your starting point for parameter tuning to enhance your PostgreSQL database's efficiency, stability, and speed.

The tuning process encompasses carefully adjusting various parameters, including memory allocation, disk I/O settings, and managing concurrent connections, all tailored to align with the specific characteristics of your hardware and operational requirements.

PostgreSQL Parameters: Scope and Priority Users Should Know

By Jobin Augustine

Jobin is a PostgreSQL Escalation Specialist in Support at Percona and open source advocate with more than 21 years of experience as a consultant, architect, administrator, writer, and trainer in PostgreSQL, Oracle, and other database technologies. He has always participated actively in the open source communities, and his main focus area is database performance and optimization.

PostgreSQL allows its users to set parameters at different scopes, and the same parameter can be specified at different places and using different methods. And there could be conflicts. Someone might be wondering why certain changes are not coming into effect, so it is important to understand/recollect the scope and priority of settings.

Here, I list the options available for users in increasing order of priority. The purpose is to give a high-level view to users.

1. Compile time parameter settings

These are the set of parameters that are set at the time of compilation. This acts as the default value for PostgreSQL. We can check these values in the boot_val field of pg_settings.

```
select name,boot_val from pg_settings;
```

These compile time settings have the least priority and can be overridden in any other levels. However, some of these parameters cannot be modified by any other means. Changing these values at compile time is not intended for common use. If a PostgreSQL user wants to change these values, they need to recompile the PostgreSQL from the source code. Some are exposed through the `configure` command line option.

Some such configuration options are:
`--with-blocksize=<BLOCKSIZE>` This sets table block size in kB. The default is 8kb.

`--with-segsize=<SEGSIZE>` This sets table segment size in GB. The default is 1GB. This means PostgreSQL creates a new file in the data directory as the table size exceeds 1GB.

`--with-wal-blocksize=<BLOCKSIZE>` sets WAL block size in kB, and the default is 8kB.

Most of the parameters have compile time defaults. That is why we can start running PostgreSQL by specifying a very minimal number of parameter values.

2. Data directory/initialization-specific parameter settings

Parameters can also be specified at the data directory initialization time. Some of these parameters cannot be changed by other means or are difficult to change.

For example, the wal_segment_size, which determines the WAL segment file, is such a parameter. PostgreSQL generates WAL segment files of 16MB by default, which can be specified only at initialization. This is the level at which decisions on whether to use data_checksums need to be taken. This can be changed later using the pg_checksums utility, but that will be a painful exercise on a big database.

The **default character encoding and locale settings** can be specified at this level. But this can also be specified at the subsequent levels. You may refer to **initdb** options for more information: https://www.postgresql.org/docs/current/app-initdb.html.

Those parameters taken from the specific data directory initialization, which overrides the built-in parameters, can be checked like this:

```
select name,setting from pg_settings where source='override';
```

This override includes some of the calculated auto-tune values for that environment.

3. PostgreSQL parameters set by environment variables

PostgreSQL executables, including the postmaster, are honoring many environment variables. But they are generally used by client tools. The most common parameter used by the PostgreSQL server (postmaster) will be **PGDATA**, which sets the parameter **data_directory**. These parameters can be specified by the service managers like **systemd**.

```
$ cat /usr/lib/systemd/system/postgresql-14.service
...
Environment=PGDATA=/var/lib/pgsql/14/data/
…
Environment=PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
Environment=PG_OOM_ADJUST_VALUE=0
```

For many automation/scripting, this will be handy. Here is an example:

```
$ export PGDATA=/home/postgres/data
$ export PGPORT=5434
$ pg_ctl start
waiting for server to start....2023-08-04 06:53:09.637 UTC [5787] LOG:  starting
PostgreSQL 15.3 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.8.5 20150623 (Red Hat
4.8.5-44), 64-bit
2023-08-04 06:53:09.637 UTC [5787] LOG:  listening on IPv6 address “::1”, port 5434
2023-08-04 06:53:09.637 UTC [5787] LOG:  listening on IPv4 address “127.0.0.1”, port 5434
2023-08-04 06:53:09.639 UTC [5787] LOG:  listening on Unix socket “/tmp/.s.PGSQL.5434”
```

As we can see, PostgreSQL took the port as 5434.

4. Configuration files

Probably, this is the method every novice user will be aware of. The fundamental configuration file is postgresql.conf, and it is the most common place to have global settings. PostgreSQL looks for a configuration file in the PGDATA by default, but an alternate location can be specified using the command line parameter config_file of postmaster. The parameter specifications can be split into multiple files and directories because Postgresql supports include and include_dir directives in the configuration files. So, there can be nested/cascaded configurations.

PostgreSQL rereads all its configuration files if it receives a SIGHUP signal. If the same parameter is set in multiple locations, the last to read will be will be considered. Among all configuration files, **postgresql.auto.conf** gets the highest priority because that is the file to read the last. That is where all “ALTER SYSTEM SET/RESET” commands keep the information.

5. Command line argument to postmaster

The postmaster, aka Postgres, has a feature to set parameters as command-line arguments (it has features to get the values also). This is one of the most reliable methods used by many of the external tools to manage PostgreSQL service. For example, the high availability solution Patroni passes some of the most critical parameters as a command line argument. Here is how the Postgres process with command-line options looks in a Patroni environment:

```
/usr/pgsql-14/bin/postgres -D /var/lib/pgsql/14/data
--config-file=/var/lib/pgsql/14/data/postgresql.conf
--listen_addresses=0.0.0.0 --port=5432
--cluster_name=kc_primary_cluster --wal_level=replica
--hot_standby=on --max_connections=100 --max_wal_senders=5
--max_prepared_transactions=0
--max_locks_per_transaction=64 --track_commit_timestamp=off
--max_replication_slots=10 --max_worker_processes=8
--wal_log_hints=on
```

So Patroni can ensure that there is no local configuration mistake that can adversely affect the availability and stability of the cluster. However, changing this is possible only at the server startup. Obviously, this has higher precedence over the values from the configuration files. The scope will be at the instance level. This answers many of the Patroni user's questions on why they cannot directly change some of the parameters in the parameter file. PostgreSQL users can check those parameters which came as command line arguments like:

```
postgres=# select name,setting from pg_settings where source='command line';
            name               |  setting
-------------------------------+---------------------
cluster_name                   | perconapg_cluster
hot_standby                    | on
listen_addresses               | 0.0.0.0
max_connections                | 100
max_locks_per_transaction      | 64
max_prepared_transactions      | 0
max_replication_slots          | 10
max_wal_senders                | 5
max_worker_processes           | 8
port                           | 5432
track_commit_timestamp         | off
wal_level                      | replica
wal_log_hints                  | on
(13 rows)
```

Parameters specifications up to this level can have "postmaster" **context**. The concept of "context" is discussed in the next section.

6. Database level setting

All options discussed so far have a global scope. Meaning they are applicable for the entire instance. But there could be reasons why a PostgreSQL user wants to change that at a specific database level. For example, one of the databases might be handling an OLTP workload where query parallelism may not be really needed and may have an adverse impact. But another database might be an OLAP system.

```
postgres=# ALTER DATABASE newdb SET
max_parallel_workers_per_gather = 4;
ALTER DATABASE
```

The concept of context

At this stage, we should recollect another concept called the "context" of a parameter. For example, the network port at which PostgreSQL listens cannot be changed at the individual database level. A change of such parameters requires a PostgreSQL restart. So, we say that the context of parameter " port" is postmaster. A change to such parameters requires the **postmaster** — the main process of the PostgreSQL — to restart.

Please refer to the documentation: https://www.postgresql.org/docs/current/view-pg-settings.html to understand different contexts of PostgreSQL parameters. We won't be allowed to change parameters at this level onwards; any attempt will be prevented.

```
postgres=# ALTER DATABASE db1 SET max_connections=100;
ERROR:  parameter "max_connections" cannot be changed without restarting the server
```

The max_connections is something the postmaster specifies at the global (instance) level, requiring a restart.

Other sets of parameters need to be communicated through postmaster only, even though they can be changed without restarting the server. That context is called sighup. Because we can signal the postmaster, and it will re-read such parameters and propagate the same to all its child processes, changing them at the database level will be prevented.

```
postgres=# ALTER DATABASE db1 SET log_filename='postgresql-DB1.log';
ERROR:  parameter "log_filename" cannot be changed now
```

You may even consider looking at the PostgreSQL source code: https://github.com/postgres/postgres/blob/6fde2d9a005a5bc04aa059d3faeb865c8dd322ce/src/backend/utils/misc/guc.c#L3376 for a much deeper understanding of the logic of "context" and what is allowed in which level.

7. User-level settings

Each user can have their preferred parameter settings so that all sessions created by that user will have that setting in place. Please remember that this user-level setting has a higher preference than database-level settings. Users can check their user-level settings like this:

```
select name,setting,source,context from pg_settings where  source='user';
```

8. Database – user combination

PostgreSQL allows us to have parameter settings that are applicable when a particular user/role connects to a particular database.

For example:

```
ALTER USER admin IN DATABASE db1 SET
max_parallel_workers_per_gather=6;
```

Setting at this level has even higher priority than everything mentioned before.

```
select name,setting,source,context from pg_settings where
name='max_parallel_workers_per_gather';

            name                  | setting |    source     | context
----------------------------------+---------+---------------+---------
 max_parallel_workers_per_gather | 6       | database user | user
(1 row)
```

9. Parameters by the client connection request

There is an option to specify parameters while making a new connection. It can be passed to PostgreSQL as part of the connection string.

For example, I want to connect to the database to perform some bulk data loading and manipulation (ETL), and I don't want to wait for any WAL writing. If there is any crash in between, I am okay with performing the ETL again. So, I am going to request a connection with synchronous_commit **off**.

```
$ psql "host=localhost user=postgres options='-c synchronous_commit=off'"
psql (14.8)
Type "help" for help.

postgres=# select name,setting,source,context from pg_settings where  name='synchronous_
commit';
      name          | setting   | source | context
--------------------+-----------+--------+---------
 synchronous_commit | off       | client | user
(1 row)
```

10. Session-level setting

Each session can decide on the settings for that session at that point in time or execution. The sessions are allowed to modify this session-level setting as and when required.

```
postgres=# set jit=off;
SET
postgres=# select name,setting,source,context from pg_settings where  name='jit';
 name | setting | source  | context
------+---------+---------+---------
 jit  | off     | session | user
(1 row)
```

A good use case is that, suppose we are going to rebuild a big index. We know that it is going to use considerable **maintenance_work_mem**. Setting this at the session level simplifies our life without affecting other sessions.

```
set maintenance_work_mem = '4GB';
```

11. Transaction-level settings

PostgreSQL allows us to specify parameters at a very small scope, like transactions. Here is an example of discouraging sequential scans in a particular transaction block.

```
postgres=# BEGIN;
BEGIN
postgres=*# SET LOCAL enable_seqscan=off;
SET
```

I prefer transaction-level settings because the changes are very local to the transaction and will be reverted once the transaction is completed. This is the most preferred place to set the work_mem to minimize the impact.

12. Object-level settings

PostgreSQL allows us to specify the parameter specific to a program block, like a PL/pgSQL function. So, the setting goes as part of the function definition.
Here is an example of the function definition to test the function-level settings.

```
CREATE OR REPLACE FUNCTION checkParams()
 RETURNS BOOLEAN
as $$
DECLARE
  nm TEXT;
  setng TEXT;
  cntxt TEXT;
  src TEXT;
BEGIN
SELECT name,setting,context,source INTO nm,setng,cntxt,src from pg_settings where
name='enable_partitionwise_join';
RAISE NOTICE 'Parameter Name: % value:%  Context:%  Source:%',nm,setng,cntxt,src;
RETURN true;
END;
$$ LANGUAGE plpgsql
SET enable_partitionwise_join = 'on'
SET enable_seqscan = 'off';
```

PostgreSQL parameters summary

PostgreSQL parameter specification is very flexible and powerful, so understanding the scope, context, and priority is important for every user. A rule of thumb could be the broader the scope, the lower the priority.

It’s All About Replication Lag in PostgreSQL

By Naveed Shaikh

Naveed has more than 11 years of experience working in a core PostgreSQL environment and is a PostgreSQL DBA II in the Managed Services department at Percona.

PostgreSQL is a popular open source relational database management system that is widely used for storing and managing data. One of the common issues that can be encountered in PostgreSQL is replication lag.

Here, we will discuss replication lag, why it occurs, and how to mitigate it in PostgreSQL.

What is replication lag?

Replication lag is the time between when data is written to the primary database and when it is replicated to the standby databases. In PostgreSQL, replication lag can occur due to various reasons such as network latency, slow disk I/O, long-running transactions, etc.

Replication lag can have serious consequences in high-availability systems where standby databases are used for failover. If the replication lag is too high, it can result in data loss when failover occurs.

The most common approach is to run a query referencing this view in the primary node.

```
postgres=# SELECT pid,application_name,client_addr,client_hostname,state,sync_state,replay_lag
postgres-# FROM pg_stat_replication
postgres-# ;
pid | application_name | client_addr | client_hostname | state | sync_state | replay_lag
-----+------------------+-------------+-----------------+-------+------------+------------
(0 rows)
postgres=#
```

Queries to check in the Standby node:

```
postgres=# select pg_is_in_recovery(),pg_is_wal_replay_paused(), pg_last_wal_receive_lsn(), pg_last_wal_replay_lsn(), pg_last_
xact_replay_timestamp();

pg_is_in_recovery  | pg_is_wal_replay_paused | pg_last_wal_receive_lsn | pg_last_wal_replay_lsn | pg_last_xact_replay_timestamp
-------------------+-------------------------+-------------------------+------------------------+-------------------------------
t                  | f                       | D1/8000000              | D1/8000000             | 2023-03-16 11:37:57.861711+00

(1 row)
postgres=#
```

Why does replication lag occur?

Replication lag can occur due to various reasons, such as:

Network latency: Network latency is the delay caused by the time it takes for data to travel between the primary and standby databases.

Various factors, such as the distance between the databases, network congestion, etc., can cause this delay:

Slow disk I/O: Slow disk I/O can be caused by various factors such as disk fragmentation, insufficient disk space, etc. Slow disk I/O can delay writing data to the standby databases.

Long-running transactions: Long-running transactions can cause replication lag because the changes made by these transactions are not replicated until the transaction is committed.

A poor configuration, like setting low numbers of max_wal_senders while processing huge numbers of transaction requests.
Sometimes, the server recycles old WAL segments before the backup can finish and cannot find the WAL segment from the primary.

Usually, this is also due to the checkpointing behavior where WAL segments are rotated or recycled.

Mitigating replication lag in PostgreSQL

There are several ways to mitigate replication lag in PostgreSQL, such as:

Increasing the network bandwidth: Increasing the network bandwidth between the primary and standby databases can help reduce replication lag caused by network latency.

Using asynchronous replication: Asynchronous replication can help reduce replication lag by allowing the standby databases to lag behind the primary database.

This means the standby databases do not have to wait for the primary database to commit transactions before replicating the data.

Tuning PostgreSQL configuration parameters: Tuning the PostgreSQL configuration parameters, such as wal_buffers, max_wal_senders, etc., can help improve replication performance and reduce replication lag.

Monitoring replication lag: Monitoring replication lag can help identify the cause of the lag and take appropriate actions to mitigate it.

PostgreSQL provides several tools for monitoring replication lag, such as pg_stat_replication, pg_wal_receiver_stats, etc.

Conclusion

Replication lag is a common issue in PostgreSQL that can seriously affect high-availability systems.

Understanding the causes of replication lag and taking appropriate measures to mitigate it can help ensure the availability and reliability of the database system.

By increasing network bandwidth, using asynchronous replication, tuning PostgreSQL configuration parameters, and monitoring replication lag, administrators can mitigate replication lag and ensure a more stable and reliable database environment.

Optimizing Database Parameters for Maximum Efficiency

By Ibrar Ahmed

Ibrar worked as a Senior Database Architect at Percona from 2018 to 2023. Before joining Percona, he was a Senior Database Architect at EnterpriseDB for 10 Years. Ibrar has 18 years of software development experience and has authored multiple books on PostgreSQL.

Out of the box, the default PostgreSQL configuration is not tuned for any particular workload. Default values are set to ensure that PostgreSQL runs everywhere, with the least resources it can consume, so it doesn't cause any vulnerabilities. It has default settings for all of the database parameters. It is primarily the responsibility of the database administrator or developer to tune PostgreSQL according to their system's workload.

What is PostgreSQL performance tuning?

PostgreSQL performance optimization aims to improve the efficiency of a PostgreSQL database system by adjusting configurations and implementing best practices to identify and resolve bottlenecks, improve query speed, and maximize database throughput and responsiveness.

Key areas include:

1. **Configuration parameter** tuning involves altering variables such as memory allocation, disk I/O settings, and concurrent connections based on specific hardware and requirements.
2. **Query optimization:** Analyzing query execution plans, identifying slow queries, and optimizing them through appropriate indexing techniques, query rewriting, or utilizing advanced features like partial indexes or materialized views.
3. **Hardware optimization:** You need to ensure that the CPU, memory, and storage components meet the performance requirements of the database workload.
4. **Statistics and monitoring:** By enabling and analyzing performance statistics and utilizing monitoring tools such as Percona Monitoring and Management, you can identify bottlenecks and track query performance over time.
5. **Index tuning:** PostgreSQL offers various types of indexes, including B-tree, hash, and generalized inverted indexes (GIN/GiST). Selecting the appropriate index type, creating composite indexes, and regularly analyzing and reindexing the database can substantially improve query speed.
6. **Schema design:** Evaluating the database schema design and making adjustments such as partitioning large tables, eliminating redundant data, and denormalizing tables for frequently accessed information can improve performance.
7. **Connection pooling:** Minimizing connection overhead and improving response times for frequently accessed data by implementing mechanisms for connection pooling and caching strategies.
8. **Load balancing and replication:** Scaling the database system horizontally by distributing the workload across multiple servers using techniques like connection pooling, read replicas, or implementing a primary-replica replication setup.
9. **Memory management:** Since PostgreSQL's performance relies heavily on efficient memory usage, monitoring and adjusting key memory-related configuration parameters like shared_buffers, work_mem, and effective_cache_size can significantly impact performance.

PostgreSQL performance optimization involves monitoring, benchmarking, and adjustments to maintain high-performing PostgreSQL databases. In addition, staying up-to-date on PostgreSQL releases and improvements can also help.

Why is PostgreSQL performance tuning important?

The performance of a PostgreSQL database has a significant impact on the overall effectiveness of an application. Here's why it's crucial for businesses to possess a high-performing database:

Responsiveness: Imagine waiting forever for the data you need to show up immediately. Nobody has time for that! Slow queries and data retrieval can lead to frustrating delays that impact the user experience. A well-performing database ensures seamless interactions, quick response times, and satisfies user expectations.

Throughput: The throughput of an application is directly influenced by the speed at which the database can process and serve queries. A high-performance database significantly decreases query execution time, empowering the application to handle more concurrent requests and deliver data faster.

Scalability: As an application expands and needs to handle more data and user loads, the database must scale accordingly. A well-performing PostgreSQL database can effectively manage increased workloads, ensuring the application remains responsive and performs well even when under heavy usage.

Efficient Resource Utilization: Enabling the effective utilization of system resources like CPU, memory, and disk I/O can optimize your PostgreSQL database while maintaining functionality. This not only results in cost sav-

ings by minimizing hardware requirements but also has the potential to decrease cloud expenses.

Data Integrity: This is crucial for ensuring a reliable and efficient database. By incorporating features like ACID compliance, transaction management, and strong error handling, the database can safeguard against data corruption or loss. This is particularly important for businesses that rely on precise and consistent data for their decision-making and operational needs.

Competitive Advantage: User retention rates and customer satisfaction can make or break any business. A high-performing database that consistently ensures excellent application performance can give businesses a competitive advantage.

Understanding the impact of queries on PostgreSQL performance

Bear in mind that while optimizing PostgreSQL server configuration improves performance, a database developer must also be diligent when writing queries for the application. Suppose queries perform full table scans where an index could be used or perform heavy joins or expensive aggregate operations. In that case, the system can still perform poorly even if the database parameters are tuned. It is essential to pay attention to performance when writing database queries.

Nevertheless, database parameters are very important, too, so let's look at the eight that have the most significant potential to improve performance.

Tuneable PostgreSQL parameters

Below are some PostgreSQL parameters that can be adjusted for improved performance based on your system and specific workload.

shared_buffer

PostgreSQL uses its own buffer and also uses kernel-buffered IO. That means data is stored in memory twice, first in the PostgreSQL buffer and then in the kernel buffer. Unlike other databases, PostgreSQL does not provide direct IO. This is called double buffering. The PostgreSQL buffer is called **shared_buffer**, which is the most effective tunable parameter for most operating systems. This parameter sets how much dedicated memory will be used by PostgreSQL for the cache.

The default value of shared_buffer is set very low, and you will not benefit much from that. It's low because certain machines and operating systems do not support higher values. But in most modern machines, you need to increase this value for optimal performance.

The recommended value is 25% of your total machine RAM. Try some lower and higher values because, in some cases, we achieve good performance with a setting over 25%. The configuration really depends on your machine and the working data set. If your working set of data can easily fit into your RAM, then you might want to increase the shared_buffer value to contain your entire database so that the whole working set of data can reside in the cache. That said, you obviously do not want to reserve all RAM for PostgreSQL.

In production environments, it is observed that a large value for shared_buffer gives excellent performance, though you should always benchmark to find the right balance.

```
testdb=# SHOW shared_buffers;
shared_buffers
----------------
128MB
(1 row)
```

Note: Be careful; some kernels **do not allow a bigger value**. Specifically, in Windows, there is no use of a higher value.

wal_buffers

PostgreSQL writes its WAL (write-ahead log) record into the buffers, which are then flushed to disk. The default size of the buffer, defined by **wal_buffers**, is 16MB, but if you have many concurrent connections, a higher value can give better performance.

effective_cache_size

The **effective_cache_size** provides an estimate of the memory available for disk caching. It is just a guideline, not the exact allocated memory or cache size. It does not allocate actual memory but tells the optimizer the amount of cache available in the kernel. If the value of this is set too low, the query planner can decide not to use some indexes, even if they'd be helpful. Therefore, setting a large value is always beneficial.

work_mem

This configuration is used for complex sorting. If you have to do complex sorting, increase the value of **work_mem** for good results. In-memory sorts are much faster than sorts spilling to disk. Setting a very high value can cause a memory bottleneck for your deployment environment because this parameter is per-user sort operation. Therefore, if you have many users trying to execute sort operations, then the system will allocate work_mem * total sort operations for all users. Setting this parameter globally can cause very high memory usage. So, it is highly recommended to modify this at the session level.

```
testdb=# SET work_mem TO "2MB";
testdb=# EXPLAIN SELECT * FROM bar ORDER BY bar.b;
                                  QUERY PLAN
------------------------------------------------------------------------------
Gather Merge  (cost=509181.84..1706542.14 rows=10000116 width=24)
Workers Planned: 4
->  Sort  (cost=508181.79..514431.86 rows=2500029 width=24)
Sort Key: b
->  Parallel Seq Scan on bar  (cost=0.00..88695.29 rows=2500029 width=24) (5 rows)

The initial query's sort node has an estimated cost of 514431.86. A cost is an arbitrary
unit of computation. For the above query, we have a work_mem of only 2MB. For testing
purposes, let's increase this to 256MB and see if there is any impact on cost.

testdb=# SET work_mem TO "256MB";
testdb=# EXPLAIN SELECT * FROM bar ORDER BY bar.b;
                                   QUERY PLAN
------------------------------------------------------------------------------
Gather Merge  (cost=355367.34..1552727.64 rows=10000116 width=24)
Workers Planned: 4
->  Sort  (cost=354367.29..360617.36 rows=2500029 width=24)
Sort Key: b
->  Parallel Seq Scan on bar  (cost=0.00..88695.29 rows=2500029 width=24)
```

The query cost is reduced to 360617.36 from 514431.86 — a 30% reduction.

maintenance_work_mem

maintenance_work_mem is a memory setting used for maintenance tasks. The default value is 64MB. Setting a large value helps in tasks like VACUUM, RESTORE, CREATE INDEX, ADD FOREIGN KEY, and ALTER TABLE.

```
postgres=# CHECKPOINT;
postgres=# SET maintenance_work_mem to '10MB';

postgres=# CREATE INDEX foo_idx ON foo (c);
CREATE INDEX
Time: 170091.371 ms (02:50.091)]

postgres=# CHECKPOINT;
postgres=# set maintenance_work_mem to '256MB';

postgres=# CREATE INDEX foo_idx ON foo (c);
CREATE INDEX
Time: 111274.903 ms (01:51.275)
```

The index creation time is 170091.371ms when maintenance_work_mem is set to only 10MB, but that is reduced to 111274.903 ms when we increase maintenance_work_mem setting to 256MB.

synchronous_commit

This is used to enforce that the commit will wait for WAL to be written on disk before returning a success status to the client. This is a trade-off between performance and reliability. If your application is designed such that performance is more important than reliability, then turn off **synchronous_commit**. This means there will be a time gap between the success status and a guaranteed write-to-disk. In the case of a server crash, data might be lost even though the client received a success message on commit. In this case, a transaction commits very quickly because it will not wait for a WAL file to be flushed, but reliability is compromised.

checkpoint_timeout, checkpoint_completion_target

PostgreSQL writes changes into WAL. The checkpoint process flushes the data into the data files. This activity is done when CHECKPOINT occurs. This is an expensive operation and can cause a huge amount of IO. This whole process involves expensive disk read/write operations. Users can always issue CHECKPOINT whenever necessary or automate the system by PostgreSQL's parameters **checkpoint_timeout** and **checkpoint_completion_target.**

The checkpoint_timeout parameter is used to set the time between WAL checkpoints. Setting this too low decreases crash recovery time, as more data is written to disk, but it also hurts performance since every checkpoint consumes valuable system resources. The checkpoint_completion_target is the fraction of time between checkpoints for checkpoint completion. A high frequency of checkpoints can impact performance. For smooth checkpointing, **checkpoint_timeout** must be a low value. Otherwise, the OS will accumulate all the dirty pages until the ratio is met and then go for a big flush.

Query optimization

In the pursuit of peak database performance, the optimization of queries is a vital area of focus. In this chapter, our experts discuss some techniques and tools for query optimization in PostgreSQL, ensuring your database operates optimally.

Among the strategies discussed, we explore using Python and PgBouncer for query optimization, leveraging Python's versatility and PgBouncer's connection pooling capabilities to streamline query processing, reduce load, and enhance overall database responsiveness. Additionally, we take a look at the powerful insights offered by pg_stat_monitor. This tool extends beyond the capabilities of traditional monitoring solutions, providing a granular view of query performance.

Whether you're seeking to optimize individual queries or achieve broader performance gains, the strategies outlined here will help you get started.

Query Optimization with Python and PgBouncer

By Mario García

Mario worked as a Technical Evangelist at Percona until 2023. He has been an active open source contributor for over 15 years.

Here, you will learn how to install and configure PgBouncer with Python to implement a connection pool for your application.

PgBouncer

PgBouncer is a PostgreSQL connection pooler. Any target application can be connected to PgBouncer as if it were a PostgreSQL server, and PgBouncer will create a connection to the actual server, or it will reuse one of its existing connections.

The aim of PgBouncer is to lower the performance impact of opening new connections to PostgreSQL.

Installation

If you're an Ubuntu user, you can install PgBouncer from the repositories:

```
$ sudo apt install pgbouncer -y
```

If not available in the repositories, you can follow the instructions below for both Debian and Ubuntu as mentioned in the Scaleway documentation

1. Create the apt repository configuration file

```
$ sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -c
s)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'
```

2. Import the repository signing key

```
$ wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key
add -
```

```
$ wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | apt-key
add -
```

3. Update the apt package manager

```
$ sudo apt update
```

4. Install PgBouncer using apt

```
$ sudo apt install pgbouncer -y
```

Configuration

After installing PgBouncer, edit the configuration files, as stated in the Scaleway documentation.

1. Set up the PostgreSQL server details in /etc/pgbouncer/pgbouncer.ini

```
database_name = host=localhost port=5432 dbname=database_name
```

You may also want to set listen_addr to * if you want to listen to TCP connections on all addresses or set a list of IP addresses.

The default listen_port is 6432.

From this article by Abdullah Alger, the settings max_client_conn and default_pool_size, the former refer to the number of applications that will make connections, and the latter is how many server connections per database. The defaults are set at 100 and 20, respectively.

2. Edit the /etc/pgbouncer/userlist.txt file and add your PostgreSQL credentials

```
“username” “password”
```

3. Add the IP address of the PgBouncer server to the PostgreSQL pg_hba.conf file

```
host all all PGBOUNCER_IP/NETMASK trust
```

By default, PgBouncer comes with a trust authentication method. The trust method can be used in a development environment but is not recommended for production. For production, HBA authentication is recommended.

4. After configuring PgBouncer, restart both the PostgreSQL and PgBouncer services

```
sudo systemctl reload postgresql
sudo systemctl reload pgbouncer
```

Python

Requirements
Dependencies

Make sure all the dependencies are installed before creating the Python script that will generate the data for your Project. You can create a requirements.txt file with the following content:

```
tqdm
faker
psycopg2
```

Or if you're using Anaconda, create an environment.yml file:

```
name: percona
dependencies:
- python=3.10
- tqdm
- faker
- psycopg2
```

You can change the Python version as this script has been proven to work with these versions of Python: 3.7, 3.8, 3.9, 3.10, and 3.11.

Run the following command if you're using pip:

```
pip install -r requirements.txt
```

Or run the following statement to configure the project environment when using Anaconda:

```
conda env create -f environment.yml
```

Database

Now that you have the dependencies installed, you must create a database named company.
Log into PostgreSQL:

```
$ sudo su postgres
$ psql
```

Create the company database:

```
create database company;
```

And create the employee's table:

```
create table employees(
id serial primary key,
fist_name varchar(50) not null,
last_name varchar(50) not null,
job varchar(100) not null,
address varchar(200) not null,
city varchar(100) not null,
email varchar(50) not null
);
```

Inserting data

Now it's time to create the Python script that will generate the data and insert it into the database.

```
from multiprocessing import Pool, cpu_count
import psycopg2
from tqdm import tqdm
from faker import Faker2/26/24, 11:14 AM PostgreSQL: Query Optimization With Python and
PgBouncer | Percona Community
https://percona.community/blog/2023/04/25/postgresql-query-optimization-with-py-
thon-and-pgbouncer/ 9/12
fake = Faker()
num_cores = cpu_count() - 1
def insert_data(arg):
x = int(60000/num_cores)
print(x)
with psycopg2.connect(database=
“database_name”
, user=
“user”
, password=
“password”
, host=
“localhost”
, port=
“6432”) as conn:
with conn.cursor() as cursor:
for i in tqdm(range(x), desc=
“Inserting Data”):
sql =
“INSERT INTO employees (first_name, last_name, job, addres
s, city, email) VALUES (%s, %s, %s, %s, %s, %s)”
val = (fake.first_name(), fake.last_name(), fake.job(), fake.addr
ess(), fake.city(), fake.email())
cursor.execute(sql, val)
if __name__
==
“
__main__
“:
with Pool() as pool:
pool.map(insert_data, range(num_cores))
```

At first, the multiprocessing pool is created and configured to use all available CPU cores minus one. Each core will call the insert_data() function.

On each call to the function, a connection to the database will be established through the default port (6432) of PgBouncer, meaning that the application will open a number of connections equal to num_cores, a variable that contains the number of CPU cores being used.

Then, the data will be generated with Faker and inserted into the database by executing the corresponding SQL statements.

In a CPU with 16 cores, the number of records inserted into the database on each call to the function will equal 60,000 divided by 15, that is 4,000 SQL statements executed.

This way, you can modify and optimize the script by configuring a connection pool with PgBouncer.

Improve PostgreSQL Query Performance Insights with pg_stat_monitor

By Ibrar Ahmed

Ibrar worked as a Senior Database Architect at Percona from 2018 to 2023. Before joining Percona, he was a Senior Database Architect at EnterpriseDB for 10 Years. Ibrar has 18 years of software development experience and has authored multiple books on PostgreSQL.

Hamid Akhtar

Hamid has more than two decades of professional software development experience and has been involved with PostgreSQL for over a decade. He is a Senior Software Engineer (Engineering Tech Lead) at Percona.

Understanding query performance patterns is essentially the foundation for query performance tuning. It, in many ways, dictates how a database cluster evolves. And then there are obviously direct and indirect cost connotations as well.

PostgreSQL provides very detailed statistics through a number of catalog views and extensions that can be easily added to provide more detailed query statistics. With each view focused on a particular aspect, the picture almost always needs to be stitched together by combining different datasets. That requires effort, and still, the whole picture might not be complete.

The pg_stat_monitor extension attempts to provide a more holistic picture by providing much-needed query performance insights in a single view.

Some useful extensions

Currently, you may be relying on a number of extensions to understand how a query behaves, the time taken in planning and execution phases, min/max/meantime values, index hits, query plan, and client application details. Here are some extensions that you might already be very familiar with.

pg_stat_activity

This view is available by default with PostgreSQL. It provides one row per server process along with current activity and query text.

In case you'd like to learn more about it, hop over to the official PostgreSQL documentation here.

pg_stat_statements

This extension is part of the contrib packages provided with the PostgreSQL server. However, you'd have to create the extension manually. It's a query-wise aggregation of statistical data with min/max/mean/ standard deviation for execution and planning times and various useful information and query text.

You can read more about pg_stat_statements at the official PostgreSQL documentation site.

auto_explain

Another useful extension is provided by the PostgreSQL server. It dumps query plans in the server log for any query exceeding a time threshold specified by a GUC
(Grand Unified Configuration).

pg_stat_monitor

While all previously mentioned views/extensions are great in their own right, one needs to manually combine client/connection information from pg_stat_activity, statistical data from pg_stat_statements, and query plan from auto_analyze to complete the dataset to understand query performance patterns.

And that's precisely the pain that pg_stat_monitor alleviates.

The feature set has been growing over the past year, with it providing, in a single view, all performance-related information that you may need to debug a low performant query. For more information about the extension, see our GitHub repository, or for user-specific documentation, see our user guide.

Feature set

- Time Interval Grouping: Instead of supplying one set of ever-increasing counts, pg_stat_monitor computes stats for a configured number of time intervals and time buckets. This allows for much better data accuracy, especially in the case of high-resolution or unreliable networks.

- **Multi-dimensional grouping:** While pg_stat_statements groups counters by (userid, dbid, queryid), pg_stat_monitor uses a more detailed group for higher precision:
    - Bucket ID (bucket),
    - User ID (userid),
    - Database ID (dbid),
    - Query ID (queryid),
    - Client IP Address (client_ip),
    - Plan ID (planid),
    - Application Name (application_name).

This allows you to drill down into the performance of queries coming from particular client addresses and applications, which we at Percona have found to be very valuable in a number of cases.

- **Capture actual parameters in the queries:** pg_stat_monitor allows you to choose if you want to see queries with placeholders for parameters or actual query examples.

- Query plan: Each SQL is now accompanied by its actual plan that was constructed for its execution. Also, we found having query parameter values is very helpful, as you can run EXPLAIN on it or easily play with modifying the query to make it run better and make communication about the query clearer when discussing with other DBAs and application developers.

- Tables Access Statistics for a Statement: This allows us to easily identify all queries that accessed a given table. This set is at par with the information provided by the pg_stat_statements.

- Histogram: Visual representation is very helpful when it can help identify issues. With the help of the histogram function, you can now view a timing/calling data histogram in response to an SQL query. And yes, it even works in PostgreSQL.

```
SELECT * FROM histogram(0, 'F44CD1B4B33A47AF') AS a(range TEXT, freq INT, bar TEXT);
       range         | freq |              bar
---------------------+------+--------------------------------
  (0 - 3)}           |    2 | ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  (3 - 10)}          |    0 |
  (10 - 31)}         |    1 | ■■■■■■■■■■■■■
  (31 - 100)}        |    0 |
  (100 - 316)}       |    0 |
  (316 - 1000)}      |    0 |
  (1000 - 3162)}     |    0 |
  (3162 - 10000)}    |    0 |
  (10000 - 31622)}   |    0 |
  (31622 - 100000)}  |    0 |
(10 rows)
```

- Functions: It may come as a surprise, but we understand that functions may internally execute statements!!! To help ease the tracking and analysis, pg_stat_monitor now provides a column that specifically helps keep track of the top query for a statement so that you can backtrack to the originating function.

- Relation Names: Relations used in a query are available in the "relations" column in the pg_stat_ monitor view. This reduces work at your and makes analysis simpler and quicker.

- Query Types: With query classification as SELECT, INSERT, UPDATE, or DELETE, analysis becomes simpler. It's another effort reduced at your end and another simplification by pg_stat_monitor.

```
SELECT bucket, substr(query,0, 50) AS query, cmd_type FROM pg_stat_monitor WHERE elevel = 0;
 bucket |                       query                        | cmd_type
--------+----------------------------------------------------+----------
      4 | END                                                |
      4 | SELECT abalance FROM pgbench_accounts WHERE aid =  | SELECT
      4 | vacuum pgbench_branches                            |
      4 | select count(*) from pgbench_branches              | SELECT
      4 | UPDATE pgbench_accounts SET abalance = abalance +  | UPDATE
      4 | truncate pgbench_history                           |
      4 | INSERT INTO pgbench_history (tid, bid, aid, delta  | INSERT
```

- Query metadata: Google's Sqlcommenter is a useful tool that, in a way, bridges that gap between ORM libraries and understanding database performance. And we support it. So, you can now put any key value data in the comments in /* … */ syntax (see Sqlcommenter documentation for details) in your SQL statements, and the information will be parsed by pg_stat_monitor and made available in the comments column in pg_stat_monitor view.

```
CREATE EXTENSION hstore;
CREATE FUNCTION text_to_hstore(s text) RETURNS hstore AS $$
BEGIN
    RETURN hstore(s::text[]);
EXCEPTION WHEN OTHERS THEN
    RETURN NULL;
END; $$ LANGUAGE plpgsql STRICT;

SELECT 1 AS num /* { “application”, java_app, “real_ip”, 192.168.1.1} */;
 num
-----
   1
(1 row)


SELECT query, text_to_hstore(comments)->’real_ip’ AS real_ip from pg_stat_monitor;
query                                                                          |  real_ip
-------------------------------------------------------------------------------+-------------
 SELECT $1 AS num /* { “application”, psql_app, “real_ip”, 192.168.1.3) */  | 192.168.1.1
```

- Logging Error and Warning: As seen in different monitoring/statics collector tools, most tools/extensions only monitor successful queries. But in many cases, monitoring ERROR, WARNING, and LOG gives meaningful information to debug the issue. pg_stat_monitor not only monitors the ERROR/WARNINGS/LOG but also collects the statistics about these queries. In PostgreSQL queries with ERROR/WARNING, there is an error level (elevel), SQL Code (sqlcode), and an error message is attached. Pg_stat_monitor collects all this information along with its aggregates.

```
SELECT substr(query,0,50) AS query, decode_error_level(elevel) AS elevel,sqlcode, calls, substr(message,0,50) message
FROM pg_stat_monitor;
                    query                     | elevel | sqlcode | calls |                      message
----------------------------------------------+--------+---------+-------+-------------------------------------------------
 select substr(query,$1,$2) as query, decode_error |        |       0 |     1 |
 select bucket,substr(query,$1,$2),decode_error_le |        |       0 |     3 |
 select 1/0;                                  | ERROR  |     130 |     1 | division by zero
```

The pg_stat_monitor extension has evolved and has become very feature-rich. We do not doubt its usefulness for DBAs, performance engineers, application developers, and anyone who needs to look at query performance. We believe it can help save many hours and help identify unexpected query behaviors. pg_stat_monitor is available on GitHub.

Monitoring

Effective monitoring is key to ensuring your PostgreSQL database performs at its best. With insights from Percona's PostgreSQL experts, this chapter guides you through the essential strategies for understanding and fine-tuning your database system.

Central to this discussion is the utilization of Percona Monitoring and Management, a powerful tool designed to provide deep visibility into the health and performance of PostgreSQL databases. We also explore the capabilities of pg_stat_monitor, an extension that offers advanced monitoring by capturing detailed query performance data.

Whether you are looking for ways to monitor in real time, analyze long-term trends, or drill down into specific performance issues, the insights provided here will equip you with the knowledge to deploy effective monitoring solutions.

PostgreSQL Checkpoints, Buffers, and WAL Usage with Percona Monitoring and Management

By Agustín Gallego

Agustín joined Percona's Support team in December 2013 and is a Senior Support Engineer. He has previously worked as a Cambridge IT examinations Supervisor and as a Junior BI, SQL & C# developer.

We will discuss extending Percona Monitoring and Management (PMM) to get PostgreSQL metrics on checkpointing activity, internal buffers, and WAL usage. With this data, we'll better understand and tune our Postgres servers.

We'll assume there are working PostgreSQL and PMM environments set up already. For generating load, we used pgbench with the following commands:

```
shell> pgbench -i -s 100 sbtest
shell> pgbench -c 8 -j 3 -T $((60*60*4)) -N -S -P 1 sbtest
```

Creating a custom query collector

Note: This step will not be needed with soon-to-come postgresql_exporter PMM versions!

The first step is to include a new query to gather data on WAL usage because this is not yet collected by default on the latest version of PMM. However, do note that only three graphs will use these custom metrics, so if you want to try out the dashboard without doing this customization, it's also possible for you to do it (at the expense of having a bit less data).

For this, it's as easy as executing the following (in the PMM client node that is monitoring the Postgres servers):

```
cd /usr/local/percona/pmm2/collectors/custom-queries/postgresql/high-resolution/
cat<<EOF >queries-postgres-wal.yml
# ####################################################
# This query gets information related to WAL LSN numbers in bytes,
# which can be used to monitor the write workload.
# Returns a COUNTER: https://prometheus.io/docs/concepts/metric_types/
# ####################################################
pg_wal_lsn:
  query: "SELECT pg_current_wal_lsn() - '0/0' AS bytes, pg_walfile_name(pg_current_wal_lsn()) as
file, pg_current_wal_insert_lsn() - '0/0' AS insert_bytes, pg_current_wal_flush_lsn() - '0/0' AS
flush_bytes"
  metrics:
    - bytes:
        usage: "COUNTER"
        description: "WAL LSN (log sequence number) in bytes."
    - file:
        usage: "COUNTER"
        description: "WAL file name being written to."
    - insert_bytes:
        usage: "COUNTER"
        description: "WAL insert LSN (log sequence number) in bytes."
    - flush_bytes:
        usage: "COUNTER"
        description: "WAL flush LSN (log sequence number) in bytes."
EOF
chown pmm-agent:pmm-agent queries-postgres-wal.yml
chmod 660 queries-postgres-wal.yml
```

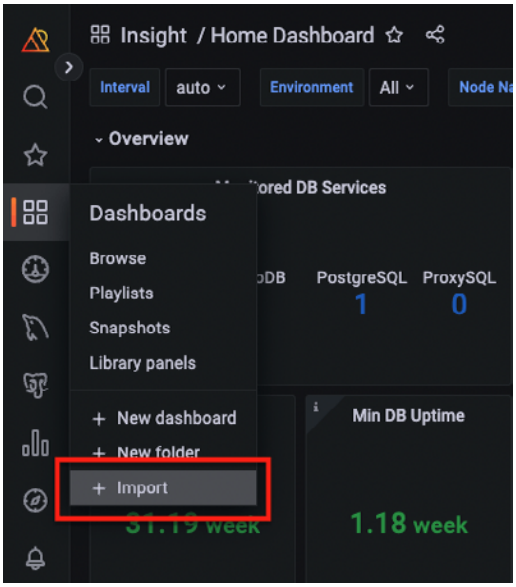
This will result in the following new metrics on PMM:

```
pg_wal_lsn_bytes
pg_wal_lsn_file
pg_wal_lsn_insert_bytes
pg_wal_lsn_flush_bytes
```

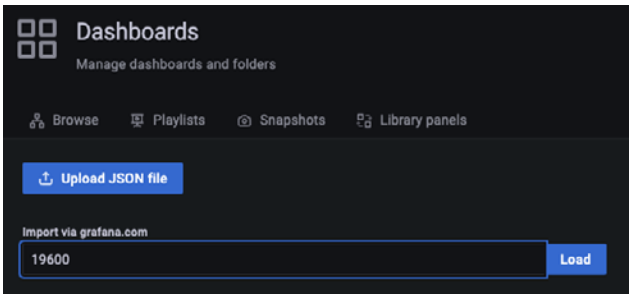
Importing the custom PMM dashboard

To view these new metrics, we will import a new custom PMM dashboard that will provide this information in a structured and easy-to-understand way. This can be done in two easy steps (since I've uploaded it to Grafana Labs).

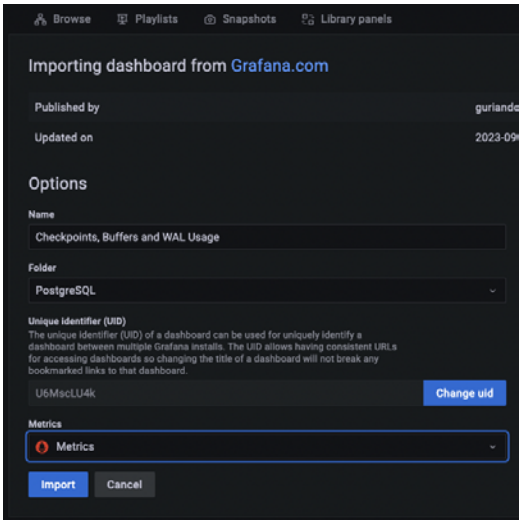
1. Click on the Dashboards -> Import menu:



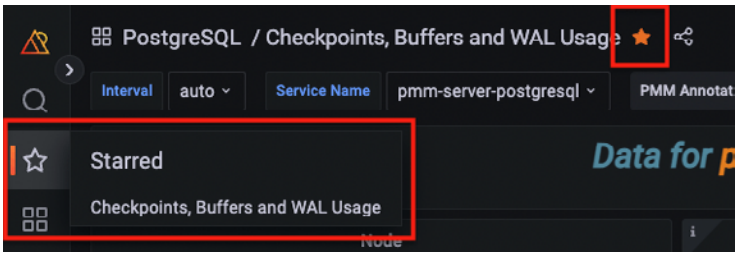
2. Import via grafana.com with ID number (19600):



3. Select the PostgreSQL folder and Metrics, and import it:



After this, you'll be taken to the dashboard, which I generally "star" for easy access later on:



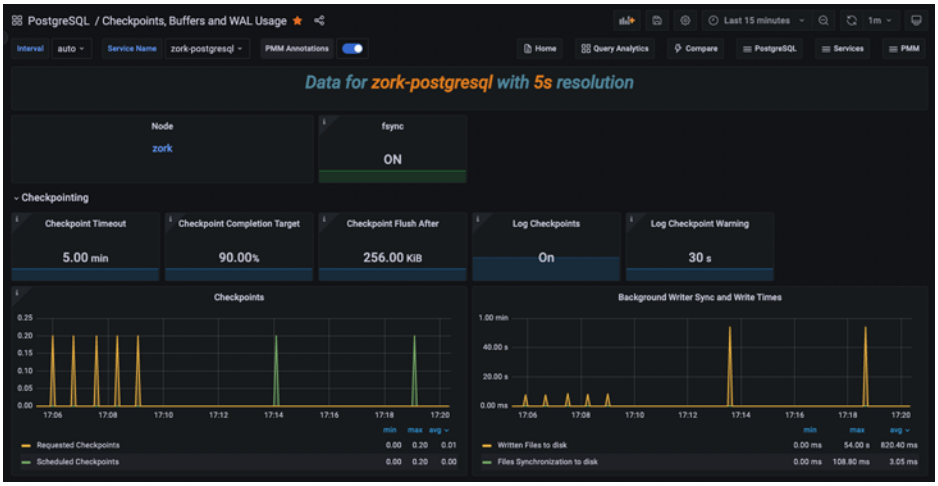
Using the new dashboard

The dashboard is divided into three main areas, with relevant configurations at the top and graphs following them. The main idea is to easily detect what the server is doing at any point in time so we can better understand how it reacts to the workload and tune accordingly. These graphs and panels show changes in usage and workload patterns and configuration changes, which can help during root cause analysis and performance reviews.

Checkpointing section

Here, we will find everything related to checkpointing: when are checkpoints started (and due to what) and when they finish. Checkpointing has a lot of impact on the write throughput/utilization by Postgres, so it's important to make sure that it's not being triggered before needed.

In the following example, we can see how at the beginning, there were many instances of forced checkpointing, not only because the checkpointer was running more often than checkpoint_timeout seconds but because of the kind of metric we can see in the Checkpoints graph (requested vs. scheduled). We can't see it yet because the WAL graphs are at the bottom, but this was fixed after we increased the value for max_wal_size.



Additionally, after the metrics refresh, we can see that the Checkpoint Timeout stat panel shows another change closer to the end of our currently selected time range:

This means that we have also changed the checkpoint_timeout value. This was done only after our workload was more stable and there were no more forced checkpoints being issued.

Buffers section

This section holds configurations and metrics related to reads to and writes from the shared buffers. Again, demonstrating it by example, let's see how much of an impact correctly sizing the shared_buffers has on our test workload. It's set at 128Mb by default, which is hardly enough for any serious workload and will mean that we should see a lot of churn in both reads and writes until we increase it. The only downside is that modifying shared_buffers needs a restart, so before doing that, we can increase bgwriter_lru_maxpages (which doesn't need a restart) to avoid the writer stopping each round and get a bit more performance out of it at the expense of I/O.

We can clearly see how, at first, there were constant reads into the shared buffers, and the background writer was doing most of the writes. Additionally, we can see the positive impact on increasing bgwriter_lru_maxpages because that process is not being throttled anymore. After we increased shared_buffers and restarted Postgres, the reads decreased to almost having to read nothing from the page cache or disks, and the writes are no longer done by the background writer but by the checkpointer.

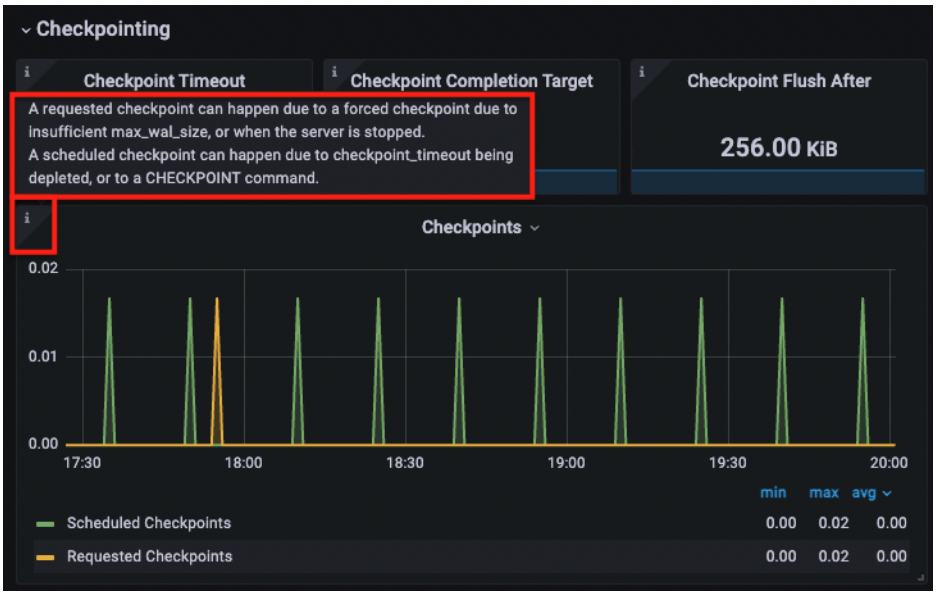
WAL usage section

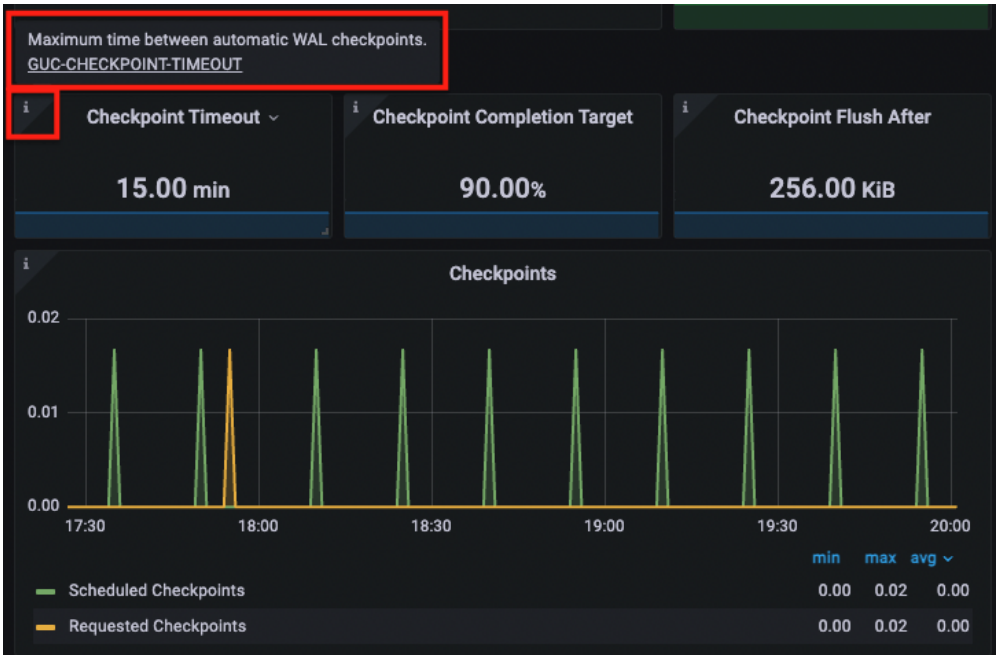
The last section pertains to write-ahead logs, another source of potential performance bottlenecks. Note that (for now) this is the only section that needs data from the custom query collector we added at the beginning, so you can use the rest of the dashboard even if you decide not to include it. PMM already collects the configurations, so we will only miss data from the graphs.

The WAL Writes Per Checkpoint Timeout graph will group the number of bytes written in chunks of checkpoint_timeout seconds so we can clearly see the expected max_wal_size value. If the chunks (shown as orange bars) exceed the max_wal_size (shown as a red line), it means that there will be forced checkpointing. We can easily dimension max_wal_size knowing that it should be larger than any orange bar, and we can tell when exactly it was changed, following changes in the red line. The WAL Writes graph uses the same metric but is shown as a rate, which can help us pinpoint heavy write times more granularly. Lastly, the WAL Locations graph is included for completeness and shows the different locations for WAL pointers (such as insertion and flush locations), which, according to the documentation, are mainly used for debugging purposes.

Hints and tips

Most graphs contain additional information and links on either the configuration variables or the metrics they show, so they are useful in case we need a bit of help interpreting them. Just hover the mouse over the "i" icon on each of them:





Getting even more data

Starting from PostgreSQL 15, we have a new view on WAL metrics: pg_stat_wal.
The metrics used in this custom collector and dashboard will work for older versions, but in the future, it will be nice to see what data we can extract from this new view and how we can use it to tune our servers.

Conclusion

This is another example of how tunable and powerful PMM is. Not only because we can add our custom metrics easily but also because it already collects many metrics we can use out of the box. Additionally, it's easy to share new PMM dashboards via the Grafana Labs page by simply publishing it and sharing the ID number. With all this new information, we can better tune our PostgreSQL instances!

Understand Your PostgreSQL Workloads Better with pg_stat_monitor

By Matt Yonkovit

Matt was the Head of Open Source Strategy at Percona until 2022, overseeing the company's community strategy and work around open source.

Percona has long been associated with pushing the limits and understanding the nuances involved in running database systems at scale, so building a tool that helps get us there brings a bit more insight and details around query performance and scale on PostgreSQL systems fits with our history. So what the hell does pg_stat_monitor do, and why should you care? Excellent question!

Currently, the de facto standard for collecting and reviewing query metrics is pg_stat_statements. This extension collects query metrics and allows you to go back and see which queries have impacted your system. Querying the extension would yield something like this:

```
postgres=# dx
List of installed extensions
-[ RECORD 1 ]-------------------------------------------------------------------
Name        | pg_stat_statements
Version     | 1.8
Schema      | public
Description | track planning and execution statistics of all SQL statements executed
-[ RECORD 2 ]-------------------------------------------------------------------
Name        | plpgsql
Version     | 1.0
Schema      | pg_catalog
Description | PL/pgSQL procedural language


postgres=# x
Expanded display is on.
postgres=# select * from pg_stat_statements;


-[ RECORD 2 ]-------+-----------------------------------------------------------
userid              | 16384
dbid                | 16608
queryid             | -7945632213382375966
query               | select ai_myid, imdb_id, year, title, json_column from movies_normalized_meta
where ai_myid = $1
plans               | 0
total_plan_time     | 0
min_plan_time       | 0
max_plan_time       | 0
mean_plan_time      | 0
stddev_plan_time    | 0
calls               | 61559
total_exec_time     | 27326.783784999938
min_exec_time       | 0.062153
max_exec_time       | 268.55287599999997
mean_exec_time      | 0.44391208084927075
stddev_exec_time    | 2.522740928486301
rows                | 61559
shared_blks_hit     | 719441
shared_blks_read    | 1031
shared_blks_dirtied | 0
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_dirtied  | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
blk_read_time       | 0
blk_write_time      | 0
wal_records         | 6
wal_fpi             | 0
wal_bytes           | 336
```

You can see here that this particular statement has been executed 61,559 times, with a total time of 27,326 Milliseconds, for a mean time of 0.44 MS.

You can also get metrics on if this statement is writing data, generating wal, etc. This is valuable to help find what statement may be missing cache and hitting the disk, or which statements may be blowing up your wal logs.

While this data is great, it could be better. Specifically, it's hard to determine if problems are worsening or improving. Also, what if that particular query that executed 61K times runs in .01ms 60K times and 1000 ms 1K times? Collecting enough data here to make better, more targeted decisions around optimization is needed. This is where pg_stat_monitor can help.

First, let me show you the output from one of the collected queries (note I am only selecting a single bucket; more on that in a second):

```
postgres=#
postgres=# x
Expanded display is on.
postgres=# select * from pg_stat_monitor ;
-[ RECORD 1 ]-------+---------
bucket              | 3
bucket_start_time   | 2022-04-27 20:13:00
userid              | movie_json_user
datname             | movie_json_test
client_ip           | 172.31.33.208
queryid             | 82650C255980E05
top_queryid         |
query               | select ai_myid, imdb_id, year, title, json_column from movies_normalized_meta where
ai_myid = $1
comments            |
planid              |
query_plan          |
top_query           |
application_name    |
relations           | {public.movies_normalized_meta}
cmd_type            | 1
cmd_type_text       | SELECT
elevel              | 0
sqlcode             |
message             |
calls               | 18636
total_exec_time     | 9022.0356
min_exec_time       | 0.055
max_exec_time       | 60.7575
mean_exec_time      | 0.4841
stddev_exec_time    | 1.568
rows_retrieved      | 18636
plans_calls         | 0
total_plan_time     | 0
min_plan_time       | 0
max_plan_time       | 0
mean_plan_time      | 0
stddev_plan_time    | 0
shared_blks_hit     | 215919
shared_blks_read    | 1
shared_blks_dirtied | 39
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_dirtied  | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
blk_read_time       | 0
blk_write_time      | 0
resp_calls          | {17946,629,55,6,0,0,0,0,0,0}
cpu_user_time       | 3168.0737
cpu_sys_time        | 1673.599
wal_records         | 9
wal_fpi             | 0
wal_bytes           | 528
state_code          | 3
state               | FINISHED
```

You can see there is a lot of extra data. Let's view these side by side:

pg_stat_monitor		PG_STAT_STATEMENTS	
bucket	3		
bucket_start_time	2022-04-27 20:13:00		
userid	movie_json_user	userid	16384
datname	movie_json_test	dbid	16608
client_ip	172.31.33.208		
queryid	82650C255980E05	queryid	-7.94563E+18
top_queryid			
query	select ai_myid, imdb_id, year, title, json_column from movies_normalized_meta where ai_myid = $1	query	select ai_myid, imdb_id, year, title, json_column from movies_normalized_meta where ai_myid = $1
comments			
planid			
query_plan			
top_query			
application_name			
relations	{public.movies_normalized_meta}		
cmd_type	1		
cmd_type_text	SELECT		
elevel	0		
sqlcode			
message			
calls	18636	calls	61559
total_exec_time	9022.0356	total_exec_time	27326.78378
min_exec_time	0.055	min_exec_time	0.062153
max_exec_time	60.7575	max_exec_time	268.552876
mean_exec_time	0.4841	mean_exec_time	0.4439120808
stddev_exec_time	1.568	stddev_exec_time	2.522740928
rows_retrieved	18636	rows	61559
plans_calls	0	plans	0
total_plan_time	0	total_plan_time	0
min_plan_time	0	min_plan_time	0
max_plan_time	0	max_plan_time	0
mean_plan_time	0	mean_plan_time	0
stddev_plan_time	0	stddev_plan_time	0
shared_blks_hit	215919	shared_blks_hit	719441
shared_blks_read	1	shared_blks_read	1031
shared_blks_dirtied	39	shared_blks_dirtied	0
shared_blks_written	0	shared_blks_written	0
local_blks_hit	0	local_blks_hit	0
local_blks_read	0	local_blks_read	0
local_blks_dirtied	0	local_blks_dirtied	0
local_blks_written	0	local_blks_written	0
temp_blks_read	0	temp_blks_read	0
temp_blks_written	0	temp_blks_written	0
blk_read_time	0	blk_read_time	0
blk_write_time	0	blk_write_time	0
resp_calls	{17946,629,55,6,0,0,0,0,0,0}		
cpu_user_time	3168.0737		
cpu_sys_time	1673.599		
wal_records	9	wal_records	6
wal_fpi	0	wal_fpi	0
wal_bytes	528	wal_bytes	336
state_code	3		
state	FINISHED		

There are 19 additional columns of collected data. Some of that extra data is used to break down the data into more granular and useful views of the data.

First up is the introduction of the concept of "buckets." What are buckets? This is a configurable slice of time. Instead of everything stored in a single big bucket, you can now add the ability to break query stats into timed buckets that allow you to look at performance changes for a query over a time period. Note these default to a max of 10 buckets, each containing 60 seconds of data (this is configurable). This means the query data is easily consumable by your favorite time-series database for even more historical analysis capabilities. We use these buckets internally to pull data into our query analytics tool and store them in a click house time-series database to provide even more analytic capabilities.

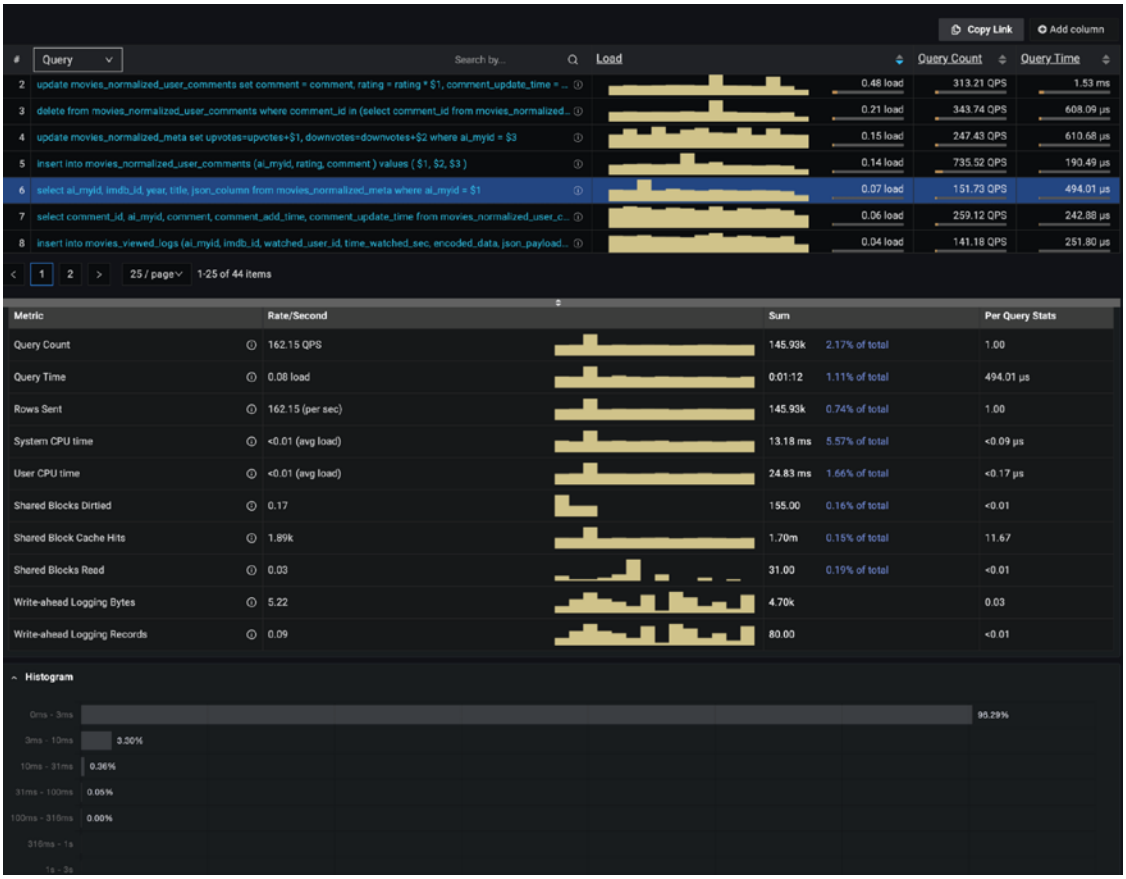
Note the difference between pg_stat_statement and pg_stat_monitor with regard to data retention. Pg_stat_monitor is best used in conjunction with another monitoring tool if you need long-term storage of query data.

Next, you will notice the inclusion of user/connection details. Many applications use the same user but have several endpoints connecting. Breaking up data via the client IP helps track down that rogue user or application server causing issues.

You can get a full breakdown of the features, settings, and columns here in the docs:

https://percona.github.io/pg_stat_monitor/REL1_0_STABLE/USER_GUIDE.html

However, I want to highlight a few of the new metrics and capabilities I am most excited about. For me, the most interesting is the ability to collect histogram data. This enables you to see if queries that deviate from the normal. One of the key things our support engineers are always looking at is how is the P99 latency, and this helps with that. You can see Percona Monitoring and Management take advantage of these features here:



With the histograms enabled, I can see and help track down where queries and performance deviate from the normal.

Additionally, you will notice the inclusion of CPU time. Why is this important? Query timings include things like waiting on disk and network resources. If you have a system with a CPU bottleneck, the queries taking the longest time may or may not be the offender.

Finally, you can configure pg_stat_monitor to store explain plans from previously run queries. This is incredibly useful when plans change over time, and you are trying to recreate what took place an hour or two ago.

Gaining additional insights and understanding your workload is critical, and pg_stat_monitor can help you do both. pg_stat_monitor enables end-to-end traceability, aggregated stats across configurable time windows, and query-wise execution time. Still, PMM visualizes this and gives the user more insight into PostgreSQL behavior.

Want to try this out for yourself? The instructions are available here: https://percona.github.io/pg_stat_monitor/REL1_0_STABLE/setup.html#installing-from-percona-repositories

Monitoring PostgreSQL Databases Using PMM

By Avinash Vallarapu

Avinash joined Percona in May 2018 and was PostgreSQL Tech Lead - Global Services until 2021. Before joining Percona, Avi worked as a Database Architect at OpenSCG for two years and as a DBA Lead at Dell for 10 Years in Database technologies.

PostgreSQL is a widely-used open source database and has been the DBMS of the year for the past two years in DB-Engine rankings. As such, there is always a need for reliable and robust monitoring solutions. While there are some commercial monitoring tools, an equally good number of open source tools are available for monitoring PostgreSQL. Percona Monitoring and Management (PMM) is one of those open source solutions that have continuous improvements and is maintained forever by Percona. It is simple to set up and easy to use.

Don't have Percona Monitoring and Management installed yet? Install it here. Here, you will see all the steps involved in monitoring PostgreSQL databases using PMM.

This is what we will be discussing:

Using the PMM docker image to create a PMM server.

- Installing PMM client on a Remote PostgreSQL server and connecting the PostgreSQL Client to PMM Server.
- Creating required users and permissions on the PostgreSQL server.
- Enabling PostgreSQL Monitoring with and without QAN (Query Analytics)

If you already know how to create a PMM Server, please skip the PMM server setup and proceed to the PostgreSQL client setup.

Using the PMM docker image to create a PMM server

PMM is a client-server architecture where clients are the PostgreSQL, MySQL, or MongoDB databases and the server is the PMM Server. We see a list of metrics on the Grafana dashboard by connecting to the PMM server on the UI. To demonstrate this setup, I have created two virtual machines, one of them the PMM Server and the second server the PostgreSQL database server.

```
192.168.80.10 is my PMM-Server
192.168.80.20 is my PG 11 Server
```

Step one:
On the PMM Server, install and start docker.

```
# yum install docker -y
# systemctl start docker
```

Here are the installation instructions for PMM Server.

Step two:
Pull the pmm-server docker image. I am using the latest PMM2 docker image for this setup.

```
$ docker pull percona/pmm-server:2
```

You see a docker image of size 1.48 GB downloaded after the above step.

```
$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
docker.io/percona/pmm-server 2 cd30e7343bb1 2 weeks ago 1.48 GB
```

Step three:
Create a container for persistent PMM data.

```
$ docker create
-v /srv
--name pmm-data
percona/pmm-server:2 /bin/true
```

Step four:
Create and launch the PMM Server. In the following step, you can see that we are binding the port 80 of the container to the port 80 of the host machine, likewise for port 443.

```
$ docker run -d
-p 80:80
-p 443:443
--volumes-from pmm-data
--name pmm-server
--restart always
percona/pmm-server:2
```

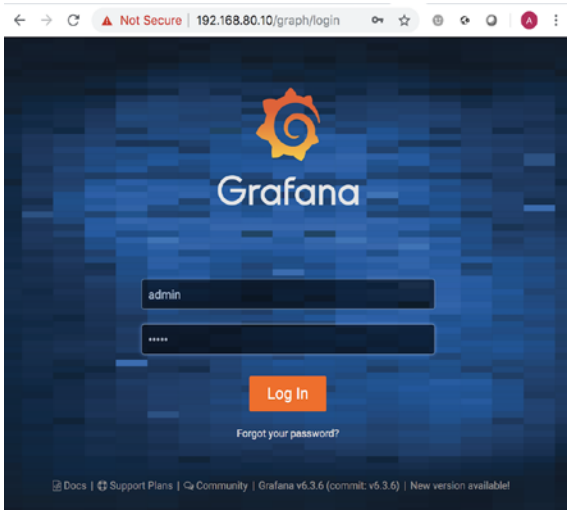
At this stage, you can modify certain settings, such as the memory you wish to allocate to the container or the CPU share, etc. You can also see more such configurable options using docker run --help. The following is just an example of how you can modify the above step with some memory or CPU allocations.

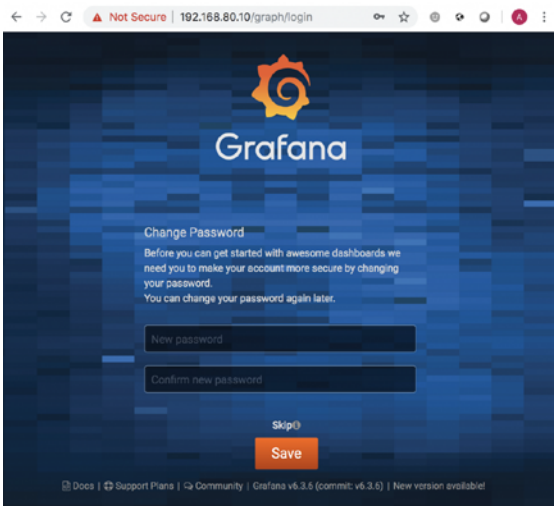
```
$ docker run -d
-p 80:80
-p 443:443
--volumes-from pmm-data
--name pmm-server
--cpu-shares 100
--memory 1024m
--restart always
percona/pmm-server:2
```

You can list the containers started for validation using docker ps.

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
bb6043082d3b percona/pmm-server:2 “/opt/entrypoint.sh” About a minute ago Up About a
minute 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp pmm-server
```

Step five:
You can now see the PMM Server Dashboard in the browser using the Host IP address. For my setup, the PMM Server's IP Address is 192.168.80.10. As soon as you put the IP in the browser, you will be asked to enter the credentials as seen in the image below. The default user and password are both: admin





Then, you will be asked to change the password or skip it.

PMM Server setup is completed after this step.

Installing PMM client on a remote PostgreSQL server

I have a PostgreSQL 11.5 Server running on 192.168.80.20. The following steps demonstrate how we can install and configure the PMM client to enable monitoring from the PMM server (192.168.80.10).

Before proceeding further, you must ensure that ports 80 and 443 are enabled on the PMM server for the PG 11 Server to connect. In order to test that, I have used telnet to validate whether ports 80 and 443 are open on the PMM Server for the pg11 server.

```
[root@pg11]$ hostname -I
192.168.80.20

[root@pg11]$ telnet 192.168.80.10 80
Trying 192.168.80.10...
Connected to 192.168.80.10.
Escape character is '^]'.

[root@pg11]$ telnet 192.168.80.10 443
Trying 192.168.80.10...
Connected to 192.168.80.10.
Escape character is '^]'.
```

Step six:
There are very few steps you need to perform on the PostgreSQL server to enable it as a client for PMM server. The first step is to install the PMM Client on the PostgreSQL Database server as follows. Based on the current PMM release, I am installing pmm2-client today. But this may change once we have a new PMM release.

```
$ sudo yum install https://repo.percona.com/yum/percona-release-latest.noarch.rpm
$ sudo yum install pmm2-client -y
```

Step seven:
The next step is to connect the client (PostgreSQL server) to the PMM Server. We could use pmm-admin config to achieve that. Following is a simple syntax that you could use in general.

```
$ pmm-admin config [<flags>] [<node-address>] [<node-type>] [<node-name>]
```

The following are the flags and other options I could use with my setup.

```
flags        : --server-insecure-tls
               --server-url=https://admin:admin@192.168.80.10:443
               (--server-url should contain the PMM Server Host information)

node-address : 192.168.80.20
               (My PostgreSQL Server)

node-type    : generic
               (As I am running my PostgreSQL database on a Virtual Machine but not on a Container,
it is generic.)

node-name    : pg-client
               (Can be any nodename you could use to uniquely identify this database server on your
PMM Server Dashboard)
```

So, the final syntax for my setup looks like the below. We can run this command as root or by using the sudo command.

Syntax: 7a

```
$ pmm-admin config --server-insecure-tls --server-url=https://admin:admin@192.168.80.10:443
192.168.80.20 generic pg-client


$ pmm-admin config --server-insecure-tls --server-url=https://admin:admin@192.168.80.10:443
192.168.80.20 generic pg-client
Checking local pmm-agent status...
pmm-agent is running.
Registering pmm-agent on PMM Server...
Registered.
Configuration file /usr/local/percona/pmm2/config/pmm-agent.yaml updated.
Reloading pmm-agent configuration...
Configuration reloaded.
Checking local pmm-agent status...
pmm-agent is running.
```

Syntax: 7b

You could also use a simple syntax such as following without node-address, node-type, node-name :

```
$ pmm-admin config --server-insecure-tls --server-url=https://admin:admin@192.168.80.10:443
```

But when you use such a simple syntax as above, node-address, node-type, node-name are defaulted to certain values. Suppose the defaults are incorrect due to your server configuration. In that case, you may better pass these details explicitly like I have done in the syntax: 7a. To validate whether the defaults are correct, you can simply use # pmm-admin config --help. In the following log, you see that the node-address defaults to 10.0.2.15, which is incorrect for my setup. It should be 192.168.80.20.

```
# pmm-admin config --help
usage: pmm-admin config [<flags>] [<node-address>] [<node-type>] [<node-name>]

Configure local pmm-agent

Flags:
  -h, --help                  Show context-sensitive help (also try --help-long and
--help-man)
      --version               Show application version
...
...
...
Args:
  [<node-address>]  Node address (autodetected default: 10.0.2.15)
```

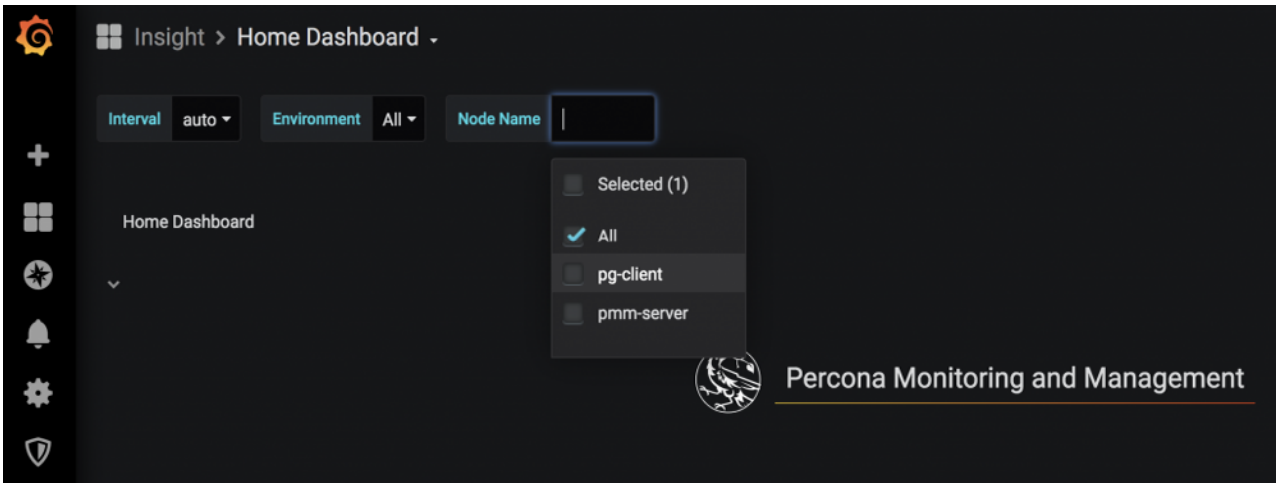
Below is an example where the default settings were perfect because I had configured my database server the right way.

```
# pmm-admin config --help
usage: pmm-admin config [<flags>] [<node-address>] [<node-type>] [<node-name>]

Configure local pmm-agent

Flags:
  -h, --help                  Show context-sensitive help (also try --help-long and
--help-man)
...
...
Args:
  [<node-address>]  Node address (autodetected default: 192.168.80.20)
  [<node-type>]     Node type, one of: generic, container (default: generic)
  [<node-name>]     Node name (autodetected default: pg-client)
```

Using steps 6 and 7a, I finished installing the PMM client on the PostgreSQL server and connected it to the PMM Server. If the above steps are successful, you should see the client listed under Nodes, as seen in the following image. Else, something went wrong.



Creating required users and permissions on the PostgreSQL server

In order to monitor your PostgreSQL server using PMM, you need to create a user *using*, which the database stats can be collected by the PMM agent. However, starting from PostgreSQL 10, you do not need to grant SUPERUSER or use SECURITY DEFINER (to avoid granting SUPERUSER). You can simply grant the role pg_monitor to a user (monitoring user).

Assuming that your PostgreSQL Version is ten or higher, you can use the following steps.

Step one:
Create a Postgres user that can be used for monitoring. You could choose any username; pmm_user in the following command is just an example.

```
$ psql -c "CREATE USER pmm_user WITH ENCRYPTED PASSWORD 'secret'"
```

Step two:
Grant pg_monitor role to the pmm_user.

```
$ psql -c “GRANT pg_monitor to pmm_user”
```

Step three:
If you are not using localhost, but using the IP address of the PostgreSQL server while enabling monitoring in the next steps, you should ensure to add appropriate entries to enable connections from the IP and the pmm_user in the pg_hba.conf file.

```
$ echo “host    all      pmm_user     192.168.80.20/32     md5” >> $PGDATA/pg_hba.conf
$ psql -c “select pg_reload_conf()”
```

In the above step, replace 192.168.80.20 with the appropriate PostgreSQL Server’s IP address.

Step four:
Validate whether you are able to connect as pmm_user to the Postgres database from the Postgres server itself.

```
# psql -h 192.168.80.20 -p 5432 -U pmm_user -d postgres
Password for user pmm_user:
psql (11.5)
Type “help” for help.
```

Enabling PostgreSQL Monitoring with and without QAN (Query Analytics)

Using PMM, we can monitor several metrics in PostgreSQL, such as database connections, locks, checkpoint stats, transactions, temp usage, etc. However, you could additionally enable Query Analytics to look at the query performance and understand the queries that need some tuning. Let us see how we can simply enable PostgreSQL monitoring with and without QAN.
Without QAN

Step one:
To start monitoring PostgreSQL, we could simply use pmm-admin to add PostgreSQL. It accepts additional arguments such as the service name and PostgreSQL address and port. As we are talking about enabling monitoring without QAN, we could use the flag: --query-source=none to disable QAN.

```
# pmm-admin add postgresql --query-source=none --username=pmm_user --password=secret
postgres 192.168.80.20:5432
PostgreSQL Service added.
Service ID  : /service_id/b2ca71cf-a2a4-48e3-9c5b-6ecd1a596aea
Service name: postgres
```

Step two:
Once you have enabled monitoring, you could validate the same using pmm-admin list.

```
# pmm-admin list
Service type  Service name         Address and port  Service ID
PostgreSQL    postgres             192.168.80.20:5432 /service_id/b2ca71cf-a2a4-48e3-
9c5b-6ecd1a596aea


Agent type                  Status     Agent ID
Service ID
pmm-agent                   connected  /agent_id/13fd2e0a-a01a-4ac2-909a-cae533eba72e
node_exporter               running    /agent_id/f6ba099c-b7ba-43dd-a3b3-f9d65394976d
postgres_exporter           running    /agent_id/1d046311-dad7-467e-b024-d2c8cb7f33c2  /
service_id/b2ca71cf-a2a4-48e3-9c5b-6ecd1a596aea
```

You can now access the PostgreSQL Dashboards and see several metrics being monitored.

With QAN

With PMM2, an additional step is needed to enable QAN. You should create a database with the same name as the monitoring user (pmm_user here). And then, you should create the extension pg_stat_statements in that database. This behavior is going to change on the next release so that you can avoid creating the database.

Step one:
Create the database with the same name as the monitoring user. Create the extension pg_stat_statements in the database.

```
$ psql -c “CREATE DATABASE pmm_user”
$ psql -c -d pmm_user “CREATE EXTENSION pg_stat_statements”
```

Step two:
If shared_preload_libraries has not been set to pg_stat_statements, we need to set it and restart PostgreSQL.

```
$ psql -c “ALTER SYSTEM SET shared_preload_libraries TO ‘pg_stat_statements’”
$ pg_ctl -D $PGDATA restart -mf
waiting for server to shut down.... done
server stopped
...
...
 done
server started
```

Step three:
In the previous steps, we used the flag: --query-source=none to disable QAN. To enable QAN, you could just remove this flag and use pmm-admin to add PostgreSQL without the flag.

```
# pmm-admin add postgresql --username=pmm_user --password=secret postgres
192.168.80.20:5432
PostgreSQL Service added.
Service ID  : /service_id/24efa8b2-02c2-4a39-8543-d5fd54314f73
Service name: postgres
```

Step four:
Once the above step is completed, you could validate the same again using pmm-admin list. But this time, you should see an additional service: qan-postgresql-pgstatements-agent.

```
# pmm-admin list
Service type  Service name         Address and port  Service ID
PostgreSQL    postgres             192.168.80.20:5432 /service_id/24efa8b2-02c2-4a39-
8543-d5fd54314f73


Agent type                   Status     Agent ID
Service ID
pmm-agent                    connected  /agent_id/13fd2e0a-a01a-4ac2-909a-cae533eba72e
node_exporter                running    /agent_id/f6ba099c-b7ba-43dd-a3b3-f9d65394976d
postgres_exporter            running    /agent_id/7039f7c4-1431-4518-9cbd-880c679513fb  /
service_id/24efa8b2-02c2-4a39-8543-d5fd54314f73
qan-postgresql-pgstatements-agent running    /agent_id/7f0c2a30-6710-4191-9373-
fec179726422  /service_id/24efa8b2-02c2-4a39-8543-d5fd54314f73
```

After this step, you can see the Queries and their statistics captured on the Query Analytics Dashboard.

Indexing

PostgreSQL indexing is a feature that allows PostgreSQL to quickly retrieve data without having to scan every row in a table. Indexes are created on one or more columns of a table and provide a fast path to locate rows matching a query condition. They are essential for optimizing search operations, especially in large databases, by reducing the amount of data that needs to be examined.

When used correctly, indexing can improve database performance by speeding up query responses. However, if not carefully managed, it can also degrade performance.

Drawing upon the expertise of our PostgreSQL professionals, this chapter highlights the dual nature of indexes: their ability to significantly enhance database efficiency and the scenarios where they might prove detrimental.

PostgreSQL Indexes Can Hurt You: Negative Effects and the Costs Involved

By Jobin Augustine

Jobin is a PostgreSQL Escalation Specialist in Support at Percona and open source advocate with more than 21 years of experience as a consultant, architect, administrator, writer, and trainer in PostgreSQL, Oracle, and other database technologies. He has always participated actively in the open source communities, and his main focus area is database performance and optimization.

Indexes are generally considered to be the panacea when it comes to SQL performance tuning, and PostgreSQL supports different types of indexes catering to different use cases. I keep seeing many articles and talks on "tuning" discussing how creating new indexes speeds up SQL, but rarely does anyone discuss removing them. The urge to create more and more indexes is found to be causing severe damage in many systems. Many times, removing indexes is what we should be doing first before considering any new indexes for the benefit of the entire system. Surprised? Understanding the consequences and overhead of indexes can help to make an informed decision and potentially save the system from many potential problems.

At a very basic level, we should remember that indexes are not free of cost. The benefits come with a cost in terms of performance and resource consumption. The following is the list of ten problems/overheads that the excessive use of indexes can cause. This post is about PostgreSQL, but most of the problems also apply to other database systems.

1. Indexes penalize the transactions

We might see an improvement in the performance of a SELECT statement after adding an index. But we should not forget that the performance gains come with a cost to the transactions on the same table. Conceptually, **every DML on a table needs to update all the indexes of the table**. Even though there are a lot of optimizations for reducing the write amplification, it is a considerable overhead.

For example, assume there are five indexes on a table; every INSERT into the table will result in an INSERT of the index record on those five indexes. Logically, five index pages will also be updated. So effectively, the overhead is 5x.

2. Memory usage

Index pages must be in memory, regardless of whether any query uses them, because they need to get updated by transactions. Effectively, the memory available for pages of the table gets less. The more indexes, the more the requirement of memory for effective caching. If we don't increase the available memory, this starts hurting the entire performance of the system.

3. Random writes: Updating indexes is more costly

Unlike INSERTS new records into tables, rows are less likely to be inserted into the same page. Indexes like B-Tree indexes are known to cause more random writes.

4. Indexes need more cache than tables

Due to random writes and reads, indexes need more pages to be in the cache. Cache requirements for indexes are generally much higher than associated tables.

5. WAL generation

In addition to WAL records of the table updates, there will also be WAL records for indexes. This helps in crash recovery and replication. If you are using any wait event analysis tools/scripts like pg_gather, the overhead of the WAL generation will be clearly visible. The actual impact depends on the index type.

Database time

Wait Events and CPU info.

Event	count
WALWrite	4508
WALSync	1908
CPU	953
DataFileWrite	478
DataFileRead	404
ClientRead	231
transactionid	10
BufferContent	1
DataFileExtend	1

Wait Event Reference

This is a synthetic test case, but if WAL-related wait events appear as any of the top wait events, it is a matter of concern for a transaction system, and we should take every step to address it.

6. More and more I/O

Not only will WAL records be generated, but we will have more pages dirtied as well. As the index pages get dirtied, they must be written back to files, leading to more I/O again—the **"DataFileWrite"** wait event, as seen in the previous screenshot.

Another side effect is indexes increase the total Active-Dataset size. By "Active dataset," I mean the tables

and indexes which are frequently queried and used. As the size of the active dataset increases, the cache becomes less and less efficient. Less-effective cache results in more datafile read, so read I/O is increased. This is in addition to the read I/O required to bring the additional index pages from storage for specific queries.

Again the pg_gather report of another system with mainly select queries shows this problem. As the Active-Dataset increases, PostgreSQL has no choice but to bring the pages from storage.

Wait Events and CPU info.

Event	count
DataFileRead	4098
CPU	1879
DataFileWrite	535
ClientRead	472
WALSync	320
WALWrite	154
BufferMapping	9
DataFileSync	3
WALInsert	1
SLRUWrite	1
DataFilePrefetch	1

Wait Event Reference

A more significant percentage of “DataFileRead” sustaining for a longer duration indicates that the Active-Dataset is much bigger, which is not cachable.

7. Impact on VACUUM/AUTOVACUUM

The overhead is not only for inserting or updating index pages, as discussed in the previous points. There is overhead in maintaining it since the indexes also need cleanups of old tuple references.

I have seen cases where autovacuum workers on a single table run for a very long duration because of the size of the table and, most importantly, the excessive number of indexes on the table. In fact, it is widespread that users see their autovacuum worker is “stuck” for hours without showing any progress for a longer duration. This happens because the index cleanup by the autovacuum is the opaque stage of autovacuum and is not visible through views like pg_stat_progress_vacuum other than the vacuum phase, which is indicated as vacuuming indexes.

Indexes can get bloated and become less efficient over time. Periodic index maintenance (REINDEX) might be needed in many systems.

8. Tunnel vision while tuning

Tunnel vision is the loss of the field view. The user may be concentrating on a particular SQL statement in an attempt to “tune” and decide on creating indexes. By creating an index for tuning a query, we are shifting more system resources to that query. Then, it may give more performance to that particular statement by penalizing others.

But as we keep creating more and more indexes for tuning other queries, the resources will shift again towards other queries. This leads to a situation where the effort to tune every query penalizes every other query. Ultimately, everyone will be hurt, and only losers will be in this war. Someone trying to tune should consider how every part of the system can co-exist (maximizing business value) rather than absolute maximum performance for a particular query.

9. Greater storage requirement

I see cases where indexes take more storage than tablets almost every day.

92	26	0.0000	183181234656	**183231897600**	**206465990656**	100261777	2023-04-16 07:28:35	2023-0 15:19:2
54	1241605	0.0041	70971990016	**70991626240**	**176785973248**	173595890	2023-04-13 04:30:42	2023-0 20:01:5
77	0	0.0000	124367200256	**128788963328**	**136557428736**	173595890	2023-04-13	2023-0
52	331	0.0000	62246199296	**62263410688**	**111950528512**	173595890	2023-04-13 04:11:22	2023-0 21:21:3
79	2070793	0.0956	38551117824	**38561808384**	**92782714880**	105649045	2023-04-18 08:27:16	2023-0 08:12:5
46	26	0.0000	45129981952	**45142474752**	**91040497664**	100261777	2023-04-16 07:28:42	2023-0 19:02:5

Indexes of : 105.8GB is 1.49x of Table 71.0GB
Total : 176.8GB

This may sound too silly for those with more money to spend on storage, but we should remember that this has a cascading effect. The total database size grows to a multiple of the actual data. So obviously, backups take more time, storage, and network resources, and then the same backup can put more load on the host machine. This would also increase the time to restore a backup and recover it. Bigger databases affect many things, including more time to build standby instances.

10. Indexes are more prone to corruption

I am not just talking about rarely occurring index-related bugs like silent index corruption of PostgreSQL 14 or index corruption due to glibc collation change, which keeps popping up now and then and affects many environments even today. Over decades of working with databases, I have observed that index corruptions are reported more frequently. (I hope anyone involved in PostgreSQL for years and who has seen hundreds of cases will agree with me). As we increase the number of indexes, we increase the probability.

What should we do?

A set of critical questions should accompany new index considerations: Is it essential to have this index, or is it necessary to speed up the query at the cost of more index? Is there a way to rewrite the query to get a better performance? Is it ok to discard the small gains and live without an index?

Existing indexes also require a critical review over a period of time. All unused indexes (those indexes with **idx_scan** as zero in **pg_stat_user_indexes**) should be considered for dropping. Scripts like the one from pgexperts can help to do more analysis.

PostgreSQL 16 has one more column in **pg_stat_user_indexes / pg_stat_all_indexes** with the name **last_idx_scan,** which can tell us when was the last time the index was used (timestamp). This will help us to take a well-informed look at all the indexes in the system.

Summary

The summary in simple words: **Indexes are not cheap. There is a cost, and the cost can be manifold. Indexes are not always good, and sequential scans are not always bad, either.** My humble advice is to avoid looking for improving individual queries as the first step because it is a slippery slope. A top-down approach to tuning the system yields better results, starting from tuning the Host machine, Operating System, PostgreSQL parameter, Schema, etc. An objective "cost-benefit analysis" is important before creating an index.

Useful Queries For PostgreSQL Index Maintenance

By Ibrar Ahmed

Ibrar worked as a Senior Database Architect at Percona from 2018 to 2023. Before joining Percona, he was a Senior Database Architect at EnterpriseDB for 10 Years. Ibrar has 18 years of software development experience and has authored multiple books on PostgreSQL.

PostgreSQL has a rich set of indexing functionality, and many articles explain the syntax, usage, and value of the index. Here, I will write basic and useful queries to see the state of database indexes. People develop databases, and after some time, when there is a demand to make changes in the architecture of software, they forget to do the previous indexes' cleanup. This approach creates a mess and sometimes slows down the database because of too many indexes. Whenever we do an update or insert, the index will be updated along with the actual table; therefore, there is a need for cleanup.

A wiki page has some queries related to PostgreSQL Index Maintenance.

Before writing the queries, I want to introduce a catalog table pg_index. The table contains information about the index. This is the basic catalog table; all the index-based views use the same table.

Step one: Sometimes, you need to see how many indexes your table has. This query will show the schema-qualified table name and its index names.

```
db=# SELECT CONCAT(n.nspname,'.', c.relname) AS table,
    i.relname AS index_name FROM pg_class c
     JOIN pg_index x ON c.oid = x.indrelid
     JOIN pg_class i ON i.oid = x.indexrelid LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
  WHERE c.relkind = ANY (ARRAY['r', 't']) AND c.relname like 'pgbench_accounts';
          table          | index_name
-------------------------+-----------------------
 public.pgbench_accounts | pgbench_accounts_pkey
 public.pgbench_accounts | pgbench_accounts_index
(2 rows)
```

Step two: As we all know, an index is a performance feature, but along with that, it is also used to ensure uniqueness. But to ensure the uniqueness, we need a separate type of index called a unique index. To check whether an index is unique, pg_index has a column named "indisunique" to identify the uniqueness of the index.

```
SELECT    i.relname AS index_name,
          indisunique is_unique
FROM      pg_class c
JOIN      pg_index x ON c.oid = x.indrelid
JOIN      pg_class i ON i.oid = x.indexrelid
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE     c.relkind = ANY (ARRAY['r', 't'])
AND       c.relname LIKE 'pgbench_accounts';
       index_name        | is_unique
-------------------------+-----------
 pgbench_accounts_pkey   | t
 pgbench_accounts_index | f
(2 rows)
```

Step three: There is a pretty simple way to get the size of the index of PostgreSQL. Here is a query to list the PostgreSQL with size.

```
SELECT pg_size_pretty(pg_relation_size('pgbench_accounts_index'));
 pg_size_pretty
----------------
 132 MB
(1 row)
```

Step four: Here is a list of the indexes with total table size and size of the index, which is very useful for comparing your table size with its corresponding indexes. It's very good to know the size of your table, index, and the total size of the table.

```
SELECT    CONCAT(n.nspname,'.', c.relname) AS table,
          i.relname AS index_name, pg_size_pretty(pg_relation_size(x.indrelid)) AS table_size,
          pg_size_pretty(pg_relation_size(x.indexrelid)) AS index_size,
          pg_size_pretty(pg_total_relation_size(x.indrelid)) AS total_size FROM pg_class c
JOIN      pg_index x ON c.oid = x.indrelid
JOIN      pg_class i ON i.oid = x.indexrelid
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE     c.relkind = ANY (ARRAY['r', 't'])
AND       n.oid NOT IN (99, 11, 12375);
          table          | index_name       | table_size | index_size | total_size
-------------------------+------------------------+------------+------------+------------
 public.pgbench_tellers  | pgbench_tellers_pkey   | 88 kB      | 64 kB  | 152 kB
 public.pgbench_accounts | pgbench_accounts_pkey  | 2561 MB    | 428 MB | 3122 MB
 public.pgbench_accounts | pgbench_accounts_index | 2561 MB    | 132 MB | 3122 MB
 public.pgbench_branches | pgbench_branches_pkey  | 8192 bytes | 16 kB  | 24 kB
(4 rows)
```

pg_relation_size: Function gives the size of relation. It is used to get the size of the table/index.

pg_total_relation_size: This is a special function that gives the total size of the table along with its all indexes.

Step five: Get the query of the index. This query will show the index creation query.

```
SELECT pg_get_indexdef(indexrelid) AS index_query
FROM   pg_index WHERE  indrelid = 'pgbench_accounts'::regclass;
                                    index_query
-------------------------------------------------------------------------------------------
CREATE UNIQUE INDEX pgbench_accounts_pkey ON public.pgbench_accounts USING btree (aid)
CREATE INDEX pgbench_accounts_index ON public.pgbench_accounts USING btree (bid)
CREATE INDEX pgbench_accounts_index_dup ON public.pgbench_accounts USING btree (bid)
(3 rows)
```

Step six: If your index becomes corrupted or bloated, you need to build that index again. At the same time, you don't want to block the operation on your table, so this REINDEX CONCURRENTLY command is your choice.

```
REINDEX INDEX CONCURRENTLY idx;
REINDEX
```

Step seven: PostgreSQL has many index methods like BTree, Hash, BRIN, GIST, and GIN. Sometimes, we want to create some specific index on a column but cannot do that. PostgreSQL has limitations; some indexes cannot be created on some data types and operators, which makes sense, too. For example, the Hash index can only be used for equal operators. Here is a query to get the list of the supported data types for a particular index.

```
SELECT amname,
       opfname
FROM   pg_opfamily,
       pg_am
WHERE  opfmethod = pg_am.oid
AND    amname = 'btree';


amname | opfname
--------+--------------------
btree  | array_ops
btree  | bit_ops
btree  | bool_ops
…
```

Step eight: This query will find the unused indexes. If index_scans is 0 or close to 0, then you can drop those indexes. But be careful, as maybe those indexes are for unique purposes.

```
SELECT s.relname AS table_name,
       indexrelname AS index_name,
       i.indisunique,
       idx_scan AS index_scans
FROM   pg_catalog.pg_stat_user_indexes s,
       pg_index i
WHERE  i.indexrelid = s.indexrelid;
table_name       | index_name             | indisunique | index_scans
-----------------+------------------------+-------------+-------------
pgbench_branches | pgbench_branches_pkey | t           | 0
pgbench_tellers  | pgbench_tellers_pkey   | t           | 0
pgbench_accounts | pgbench_accounts_pkey | t           | 0
(3 rows)
```

Step nine: A query is used to find a duplicate index. In this example, pgbench_accounts has two of the same indexes. There is no need to have multiple same indexes with a different name on a table. As we already discussed, in case of update/insert, all the indexes get updated along with the actual table, which hurts the performance.

```
SELECT   indrelid::regclass table_name,
         att.attname column_name,
         amname index_method
FROM     pg_index i,
         pg_class c,
         pg_opclass o,
         pg_am a,
         pg_attribute att
WHERE    o.oid = ALL (indclass)
AND      att.attnum = ANY(i.indkey)
AND      a.oid = o.opcmethod
AND      att.attrelid = c.oid
AND      c.oid = i.indrelid
GROUP BY table_name,
         att.attname,
         indclass,
         amname, indkey
HAVING count(*) > 1;
table_name | column_name | index_method
------------+-------------+--------------
foo        | a           | btree


(1 row)
```

Conclusion

PostgreSQL has catalog tables to store the index information; therefore, we can write as many queries as needed. This shows some basic queries and shows how to use the catalog tables to write the queries.

Vaccuming

Executing the vacuum command in PostgreSQL is an important maintenance operation designed to enhance database performance and free up disk space. It's vital to avoid the buildup of redundant data, often referred to as "dead tuples" or "dead rows," which can degrade query efficiency, but its benefits are maximized only when it is correctly configured and diligently monitored.

For this chapter, we have compiled a few insights from experts into vacuum strategies, including the refinement of manual vacuuming and the automation offered by autovacuum, to provide you with the knowledge to fine-tune vacuum settings to suit your specific database needs.

PostgreSQL Vacuuming Command to Optimize Database Performance

By Ibrar Ahmed

Ibrar worked as a Senior Database Architect at Percona from 2018 to 2023. Before joining Percona, he was a Senior Database Architect at EnterpriseDB for 10 Years. Ibrar has 18 years of software development experience and has authored multiple books on PostgreSQL.

In PostgreSQL, the vacuum command is a maintenance task that helps to **optimize database performance** and reclaim disk space. Using the PostgreSQL vacuum command involves removing deleted or outdated rows from tables and indexes and updating statistics used by the query planner. This process is necessary to prevent the accumulation of unnecessary data, known as “dead tuples” or “dead rows,” which can take up significant space and slow down queries.

Multi-version concurrency control (MVCC)

To maintain consistency and prevent data loss due to concurrent updates, PostgreSQL employs multi-version concurrency control (MVCC). PostgreSQL and other database management systems use MVCC to ensure consistent reads and prevent data loss from concurrent updates. PostgreSQL is achieved by storing multiple versions of each row within the database, allowing transactions to access a consistent data snapshot.

In PostgreSQL databases, each row within a single table is assigned a transaction ID referred to as an "xmin." This ID signifies the transaction that inserted the row. When a row is updated or deleted, it is not immediately removed from the table. Instead, a new version of the row is inserted with a new transaction ID, while the old version is marked as "dead" with a transaction ID called an "xmax."

When a transaction reads a row, it utilizes the xmin and xmax values to determine whether the row is visible to the transaction. If the xmin value is greater than the transaction's "snapshot" (a record of the transaction IDs that were in progress when the transaction began), the row is not visible to the transaction. If the xmax value equals the transaction's ID, the row has been deleted by the transaction and is also not visible. In all other cases, the row is visible to the transaction.

This allows transactions to access a consistent data snapshot, as they can only see rows committed when the transaction began. It also prevents data loss due to concurrent updates, as conflicting updates result in inserting new row versions rather than overwriting the existing data.

Although MVCC incurs some overhead in terms of storage and performance due to the need to maintain multiple versions of each row, it is a crucial feature of PostgreSQL and other database systems that support concurrent updates.

This allows multiple versions of each row to be stored, enabling transactions to access a consistent data snapshot. However, this can result in the accumulation of dead tuples as rows are updated or deleted.

PostgreSQL vacuuming

The vacuum process in PostgreSQL helps maintain the database's performance and disk space efficiency by removing rows that are no longer needed.

These rows accumulate because PostgreSQL uses MVCC to allow multiple transactions to access the same data simultaneously without conflicts. During the vacuum process, tables and indexes are scanned, and these dead tuples are removed, helping to reclaim space and improve query performance. It is essential to run a vacuum periodically to keep the database running smoothly. MVCC stores multiple versions of each row, so dead tuples are not immediately removed when a row is updated or deleted.

PostgreSQL vacuum configuration parameters

Vacuum Full

The VACUUM FULL command in PostgreSQL is a tool used to reclaim disk space occupied by deleted or obsolete data. Unlike the regular VACUUM command, it performs a more thorough cleanup by physically rearranging the data on the disk. This ensures that all dead tuples are completely removed, resulting in a more compact storage structure and improved query performance.

VACUUM FULL requires exclusive access to the table, so while the command runs, other transactions cannot read from or write to the table, potentially impacting database availability. It's important to know this is a resource-intensive operation that may cause database disruptions and longer execution times.

Verbose

The term "Verbose" is used when referring to Postgres log output. Normally, PostgreSQL logs only the barest of information when it's first set up, i.e., STARTUP, SHUTDOWN, ERROR, and FATAL messages. Log output

verbosity increases substantially when the logging parameters are set to record additional information relating to client access details and Postgres background processes such as CHECKPOINTs and VACUUM. The most "verbose" setting is when each and every SQL statement is logged, which can increase the Postgres log size, generating GB of messages in a matter of a few minutes.

PostgreSQL Vacuum and Analyze

The vacuum process removes dead tuples and updates statistics used by the query planner to more accurately estimate the number of rows returned by a query and choose the most efficient execution plan. There are two types of vacuum in PostgreSQL: VACUUM and ANALYZE. VACUUM removes dead tuples and updates statistics, while ANALYZE only updates statistics. It is generally recommended to run both VACUUM and ANALYZE together.

The vacuum command can be initiated manually using SQL commands or automated using the autovacuum background process, which runs based on configurable thresholds such as the number of dead tuples or live rows in a table. In PostgreSQL 15, the vacuum process has been optimized to make it more efficient and faster to vacuum large tables. It includes improvements such as the ability to vacuum multiple partitions of the same table in parallel, vacuum indexes concurrently, and skip vacuuming indexes unaffected by an update.

From a technical perspective, the vacuum process in PostgreSQL 15 involves several components. The vacuum daemon (autovacuum) initiates vacuum operations based on configurable thresholds. The vacuum worker executes the actual vacuum operation, scanning the table or index and removing dead tuples while updating statistics.

Autovacuum

Autovacuum, which automates routine vacuum maintenance, is enabled by default in PostgreSQL and can be configured using several parameters in postgresql.conf file. PostgreSQL has several settings related to vacuum that can be configured to control how the vacuum process runs. You can find the following settings in postgresql.conf file and include:

- autovacuum: This setting enables or disables the autovacuum background process. By default, autovacuum is enabled.
- autovacuum_vacuum_threshold: This setting determines the minimum number of dead rows that must be present in a table before it is vacuumed. The default value is 50.
- autovacuum_analyze_threshold: This setting determines the minimum number of live rows that must be present in a table before it is analyzed. The default value is 50.
- autovacuum_vacuum_scale_factor: This setting is a multiplier that determines how many dead rows are needed to trigger a vacuum based on the table size. The default value is 0.2.
- autovacuum_analyze_scale_factor: This setting is a multiplier that determines how many live rows are needed to trigger an analyze based on the size of the table. The default value is 0.1.
- autovacuum_vacuum_cost_delay: This setting determines the time (in milliseconds) the autovacuum will wait before starting a vacuum operation. The default value is 20.
- autovacuum_vacuum_cost_limit: This setting determines the maximum number of rows that can be vacuumed in a single vacuum operation. The default value is 200.

Here is an example of configuring some of the vacuum parameters in the postgresql.conf file:

```
autovacuum = on
autovacuum_vacuum_threshold = 100
autovacuum_analyze_threshold = 100
autovacuum_vacuum_scale_factor = 0.5
autovacuum_analyze_scale_factor = 0.2
autovacuum_vacuum_cost_delay = 50
autovacuum_vacuum_cost_limit = 500
```

In this example, autovacuum is enabled, and the thresholds for vacuum and analyze are set to 100. The scale factors for vacuum and analyze are set to 0.5 and 0.2, respectively, which means that a vacuum will be triggered when there are 50 dead rows per 1,000 live rows in the table (0.5 x 100), and an analyze will be triggered when there are 20 live rows per 1,000 rows in the table (0.2 x 100). The vacuum cost delay is set to 50 milliseconds, and the vacuum cost limit is set to 500 rows. This means that the autovacuum will wait 50 milliseconds before starting a vacuum operation and will vacuum a maximum of 500 rows at a time.

It is important to configure these settings to ensure that vacuum and analyze properly are running effectively and not causing too much load on the database. It is also a good idea to monitor the activity of the autovacuum and manually vacuum tables that are not adequately maintained by the autovacuum.

Autovacuum can also be configured on a per-table basis using the autovacuum_vacuum_cost_delay and autovacuum_vacuum_cost_limit parameters in the table's storage parameters. These parameters control how aggressively autovacuum vacuums the table, with a lower cost delay causing the vacuum to run more frequently and a higher cost limit allowing more rows to be vacuumed at once.

Parallel vacuuming

Parallel vacuum is a feature in PostgreSQL that allows the vacuum process to be run concurrently on multiple cores or processors for the same table, improving the performance of the vacuum operation. This can be especially useful for vacuuming large tables, as it allows the vacuum process to use multiple CPUs to scan and process the table in parallel.

The parallel vacuum was introduced in PostgreSQL 13 as an experimental feature and made generally available in PostgreSQL 14. To use a parallel vacuum, you need to set the "max_parallel_workers_per_gather" parameter in the postgresql.conf configuration file to a value greater than one. To enable parallel processing, specify the "PARALLEL" option when running a vacuum or analyze command.

For example, to run a parallel vacuum on a table named "foo_table," you can use the following command:

```
VACUUM (PARALLEL, ANALYZE) foo_table;
```

You can also specify the "PARALLEL" option when running a vacuum or analyze command on an entire schema or database:

```
VACUUM (PARALLEL, ANALYZE) schema_name.*; VACUUM (PARALLEL, ANALYZE);
```

Note: Keep in mind that parallel vacuum can increase the load on the database server, as it requires multiple CPU cores to be used simultaneously. You should carefully monitor the performance of your PostgreSQL database when using a parallel vacuum and adjust the "max_parallel_workers_per_gather" parameter as needed to find the optimal setting for your workload.

Transaction wraparound

Transaction wraparound is a phenomenon that can occur in PostgreSQL when the maximum transaction ID (TXID) has been reached, and the system wraps around to reuse old transaction IDs. This can cause problems if there are still rows in the database with a higher transaction ID than the current maximum, as they will be considered “dead” and removed by the vacuum process.

To understand how transaction-wraparound works, it is essential to understand how PostgreSQL manages transaction IDs. PostgreSQL transactions are assigned a unique transaction ID called a “xid.” The xid is a 32-bit integer, meaning it has a maximum value of 2^32-1, or 4,294,967,295. When this maximum value is reached, the system wraps around and reuses old xids.

To prevent transaction wraparound from causing problems, it is important to run the vacuum process and remove dead rows regularly. You can use the following query to check for tables at risk of transaction wraparound.

Vacuum statistics

To view the vacuum history for all tables in the current schema:

This query retrieves vacuum statistics for all tables in the “public” schema in the current database. This query can help monitor the status of the vacuum process and identify tables that may need to be vacuumed or analyzed. For example, if a table has many dead rows or has not been vacuumed or analyzed recently. In that case, it may be worth running a manual vacuum or analyzing operations to improve the performance of the database.

Download our free e-book, “The 6 Common Causes of Poor Database Performance,” to see the most common database performance issues and get expert insights on how to fix them.

```
SELECT
    n.nspname as schema_name,
    c.relname as table_name,
    c.reltuples as row_count,
    c.relpages as page_count,
    s.n_dead_tup as dead_row_count,
    s.last_vacuum,
    s.last_autovacuum,
    s.last_analyze,
    s.last_autoanalyze

FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_stat_user_tables s ON s.relid = c.oid
WHERE c.relkind = ‘r’ AND n.nspname = ‘public’;

-[ RECORD 1 ]----+------------------------------
schema_name      | public
table_name       | pgbench_accounts
row_count        | 9.999965e+06
page_count       | 163935
dead_row_count   | 41705
last_vacuum      | 2022-12-25 16:00:36.231734+00
last_autovacuum  |
last_analyze     | 2022-12-25 16:00:18.90299+00
last_autoanalyze |
-[ RECORD 2 ]----+------------------------------
schema_name      | public
table_name       | pgbench_branches
row_count        | 100
page_count       | 1
dead_row_count   | 41
last_vacuum      | 2022-12-25 16:00:44.722317+00
last_autovacuum  |
last_analyze     | 2022-12-25 16:00:16.254529+00
last_autoanalyze | 2022-12-25 16:01:45.957663+00
```

To view the list of tables that have been modified since the last vacuum:

```
SELECT
    n.nspname as schema_name,
    c.relname as table_or_index_name,
    c.relkind as table_or_index,
    c.reltuples as row_count,
    s.last_vacuum,
    s.last_autovacuum,
    s.last_analyze,
    s.last_autoanalyze
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_stat_user_tables s ON s.relid = c.oid
WHERE (c.relkind = 'r' or c.relkind = 'i')
AND (s.last_vacuum < s.last_autovacuum OR s.last_vacuum < s.last_analyze);
```

To view the list of tables and indexes that have a high number of dead rows:

```
SELECT
    n.nspname as schema_name,
    c.relname as table_name,
    c.reltuples as row_count,
    s.n_dead_tup as dead_row_count

FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_stat_user_tables s ON s.relid = c.oid
WHERE c.relkind = 'r' AND s.n_dead_tup > 0;

SELECT
    n.nspname as schema_name,
    c.relname as index_name,
    s.n_dead_tup as dead_row_count

FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_stat_user_tables s ON s.relid = c.oid

    WHERE (c.relkind = ''r'' or c.relkind = ''i'')
    AND (s.last_vacuum < s.last_autovacuum OR s.last_vacuum < s.last_analyze)') AS t(
        schema_name text,
        table_or_index_name text,
        table_or_index char(1),
        row_count bigint,
        last_vacuum timestamp,
        last_autovacuum timestamp,
        last_analyze timestamp,
        last_autoanalyze timestamp
    );
```

You can write the above query for all the databases using dblink. This will give information about all the databases.

```
SELECT *
FROM dblink('host=<host> port=<port> dbname=<database> user=<username>
password=<password>',
    'SELECT
        n.nspname as schema_name,
        c.relname as table_or_index_name,
        c.relkind as table_or_index,
        c.reltuples as row_count,
        s.last_vacuum,
        s.last_autovacuum,
        s.last_analyze,
        s.last_autoanalyze
    FROM pg_class c
    JOIN pg_namespace n ON n.oid = c.relnamespace
    LEFT JOIN pg_stat_user_tables s ON s.relid = c.oid
    WHERE (c.relkind = ''r'' or c.relkind = ''i'')
    AND (s.last_vacuum < s.last_autovacuum OR s.last_vacuum < s.last_analyze)') AS t(
        schema_name text,
        table_or_index_name text,
        table_or_index char(1),
        row_count bigint,
        last_vacuum timestamp,
        last_autovacuum timestamp,
        last_analyze timestamp,
        last_autoanalyze timestamp
    );
```

Note: You need to replace the placeholders <host>, <port>, <database>, <username>, and <password> with the actual values for your server.

Conclusion

It is important to properly configure the autovacuum to avoid overloading the database with too many vacuums. It is also a good idea to monitor the activity of the autovacuum and manually vacuum tables that are not adequately maintained by the autovacuum.

In summary, PostgreSQL vacuuming is an essential maintenance task in PostgreSQL that helps to reclaim space and improve performance. It is generally recommended to run the vacuum regularly, either manually using the VACUUM SQL command or automatically using the autovacuum background process. This helps to ensure that the database remains efficient and free of dead rows and reduces the risk of transaction wraparound. Autovacuum is a convenient way to automate this process, but it is important to configure and monitor it to ensure it runs effectively and adequately.

Why Custom Scheduled Jobs for Vacuuming and Fine-grained Autovacuum Tuning Are Unavoidable

By Jobin Augustine

Jobin is a PostgreSQL Escalation Specialist in Support at Percona and open source advocate with more than 21 years of experience as a consultant, architect, administrator, writer, and trainer in PostgreSQL, Oracle, and other database technologies. He has always participated actively in the open source communities, and his main focus area is database performance and optimization.

PostgreSQL's built-in autovacuum – the housekeeper – is improving, version after version. It is becoming more capable while reducing its overhead and addressing edge cases. I think there is no PostgreSQL version that comes out without any autovacuum improvement, and there is no doubt that it is good enough for small and not performance-critical systems.

But the autovacuum is far from a perfect fit for many environments. While working with many customer environments, we keep seeing cases where the built-in logic is insufficient. As I keep doing fixes for many Percona customers, I thought of noting down important points for everyone.

Most common problems/limitations

- **Autovacuum algorithm/strategy:** DML activity is the only trigger
  There are some serious limitations to the algorithm/strategy used by autovacuum. As discussed in the blog post: Tuning Autovacuum in PostgreSQL and Autovacuum Internals, the tables become candidates for autovacuum based on parameters like

    - **autovacuum_vacuum_scale_factor**
    - **autovacuum_vacuum_threshold**
    - **autovacuum_analyze_scale_factor**
    - **autovacuum_analyze_threshold**
    - **autovacuum_vacuum_insert_scale_factor**
    - **autovacuum_vacuum_insert_threshold**

- Basically, all these parameters are based on the DML operations performed in the table. The first four parameters are based on UPDATEs and DELETEs on the table because they cause the "dead" tuples. The 5th and 6th parameters are obviously based on INSERTs.

  Considering only the DML activities for triggering autovacuum can only solve part of the problem because autovacuum has a lot more things to do than just handling dead tuples. The most important and crucial thing is the FREEZE operation / Table age. So, it is quite normal to see tables get aged until they hit the autovacuum_freeze_max_age limit, triggering the aggressive autovacuum to prevent the wraparound. Subsequently, they are heavy on the system and cause other side effects at unexpected times, which is not what we want to see in production systems.

- **Candidate tables are ordered alphabetically.** This is not something very widely discussed. This has another serious side effect: few tables always get priority, and few others never get a chance to get vacuumed. They starve until they reach the limit autovacuum_freeze_max_age. Again, the aggressive autovacuum kicks in with aggression and side effects.

- **Tables become candidates for autovacuum during peak hours.** The autovacuum settings are based on scale factors/thresholds and Indirectly based on DML Operations. The chance that the table crosses these limits is high when there is a high number of transactions on the table – which is the peak hours. Effectively, it gets kicked in at the very wrong time.

- **No consideration for long starving tables.** It is very common to see a few tables become candidates for vacuuming too frequently and occupy all workers repeatedly. Meanwhile, other tables down the list of candidature remain unvacumed for a long time. There is no way to give some weightage for tables which are not considered for long.

- **No way to control the throttle of autovacuum workers dynamically.** This is probably the worst, even if an informed DBA wants to adjust the autovacuum_vacuum_cost_limit based on need or time window and signal the PostgreSQL.

For example:

```
ALTER SYSTEM set autovacuum_vacuum_cost_limit = 2000;
select pg_reload_conf();
```

- This has no effect on the currently running autovacuum workers. Only the next worker starting will consider this setting. So this can not be used for addressing the problem.

- **The attempt by DBAs to tune parameters often backfires.** After seeing the aged tables and starving tables, desperate DBAs keep aggressive settings and a higher number of workers. Many times, this pushes the system way beyond its limit because everything gets in the wrong time with high aggression when the system has already a high number of active sessions. Multiplied by the maintenance_work_mem allocations by workers. System performance suffers to a great extent. The worst I have seen is

autovacuum workers occupying up to 50% of server resources.

- **Autovacuum during the active time window defeats its own purpose.** The autovacuum worker will be referring to an old xid/snapshot if it takes time to complete during the high activity window. So effectively, it won't be cleaning the dead tuples generated during the same duration, which is against the very purpose of autovacuum

- **Many Starved tables reach the wraparound prevention condition at the same time.** It is very common to see that the tables that are starved for a longer duration of autovacuum reach autovacuum_freeze_max_age and wraparound prevention aggressive vacuum get triggered concurrently, making the situation much worse.

Due to such ineffectiveness, we keep seeing DBAs tending to disable the autovacuum altogether and invite a bigger set of problems and even outages. At the very least, my request to anyone who is new to PostgreSQL is to please never try to turn off the autovacuum. That's not the way to address autovacuum-related issues.

Tuning Autovacuum

Tuning autovacuum is obviously the first line of action. Autovacuum settings can be adjusted at the global level or at the table level.

Global level settings

The parameters autovacuum_vacuum_cost_limit, and autovacuum_vacuum_cost_delay are the main two parameters to control the throttle of the autovacuum workers.

autovacuum_max_workers controls how many workers will be working at a time on different tables. By default, autovacuum_vacuum_cost_limit will be disabled (-1), which means the value of the other parameter vacuum_cost_limit will be in effect. So, the very first thing suggested is to set a value for autovacuum_vacuum_cost_limit, which will help us control the autovacuum workers alone.

One common mistake I see across many installations is that autovacuum_max_workers is set to a very high value, like 15, assuming that this makes the autovacuum run faster. Please remember that autovacuum_vacuum_cost_limit is divided among all workers. So the higher the number of workers, each worker runs slower. And slower workers mean ineffective cleanup as mentioned above. Moreover, each of them can occupy up to maintenance_work_mem. In general, the default value of autovacuum_max_workers, which is 3, will be sufficient. Please consider increasing it only if it is an absolute necessity.

Table-level settings

The blanket setting at the Instance level might not work great for at least a few tables. These outliers need special treatment, and tuning the settings at table level might become unavoidable. I would start with those tables which become candidates too frequently for autovacuum.

PostgreSQL with a log_autovacuum_min_duration setting gives great details of those tables that are frequently becoming candidates, and those autovacuum runs that took considerable time and effort. Personally, I prefer this as the starting point. A summary of autovacuum runs can be obtained by comparing the autovacuum_count of pg_stat_all_tables taken in two different timestamps also. We need to consider the HOT (Heap Only Tuple) updates and fillfactor. Hot update information can be analyzed using the n_tup_hot_upd of the same view (pg_stat_all_tables). Tuning this can bring down the vacuum requirements drastically. Equipped with all this information analysis, specific table-level settings can be adjusted. For example:

```
alter table t1 set (autovacuum_vacuum_scale_factor=0.0, autovacuum_vacuum_
threshold=130000, autovacuum_analyze_scale_factor=0.0, autovacuum_analyze_
threshold=630000, autovacuum_enabled=true, fillfactor=82);
```

Tools/scripts like pg_gather can be handy to assess the autovacuum frequency and suggest settings like fillfactor and table level autovacuum settings.

Supplementary scheduled vacuum job

Our aim is not to disable the autovacuum but to supplement the autovacuum with our knowledge about the system. It need not be complex at all. The simplest we can have is to run a 'VACUUM FREEZE' on tables which are having maximum age for itself or its TOAST.

For example, we can have vaccumjob.sql file with the following content:

```
set ECHO all
WITH cur_vaccs AS (SELECT split_part(split_part(substring(query from '.*..*'),'.',2),' ',1) as tab
FROM pg_stat_activity WHERE query like 'autovacuum%')
SELECT 'VACUUM (FREEZE,ANALYZE) "'|| n.nspname ||'"."'|| c.relname ||'";'
 FROM pg_class c
 JOIN pg_namespace n ON c.relnamespace = n.oid
 LEFT JOIN pg_class t ON c.reltoastrelid = t.oid and t.relkind = 't'
 WHERE c.relkind in ('r','m') AND NOT EXISTS (SELECT * FROM cur_vaccs WHERE tab = c.relname)
ORDER BY GREATEST(age(c.relfrozenxid),age(t.relfrozenxid)) DESC
LIMIT 100;
gexec
```

The query gets 100 aged tables that are not currently undergoing autovacuum and run a "VACUUM FREEZE" on them. (The gexec at the end executes the query output.)

This can be scheduled using corn for a low activity window like:

```
20 11 * * * /full/path/to/psql -X -f /path/to/vacuumjob.sql &gt; /tmp/vacuumjob.out 2&gt;&amp;1
```

If there are multiple low-impact windows, all of them can be made use of using multiple schedules.

A copy of the above SQL script is available on GitHub for easy download. Practically, we have seen that the supplementary, scheduled vacuum jobs, based on the table age approach, have the following positive effects.

- The chance of those tables becoming candidates again during the peak times is drastically reduced.
- Able to achieve very effective utilization of server resources during the off-peak times for the vacuum and freeze operation.
- Since the candidature was selected based on totally different criteria (age of table) than the default (scale factor and threshold), the chance of a few tables starving forever is eliminated. Moreover, that removes the possibility of the same table becoming a candidate for vacuum again and again.
- In customer/user environments, the wraparound prevention autovacuum is almost never reported again.

Summary

It is not rare to see systems where autovacuum remains untuned or poor settings are used from the instance level to the table level. Just want to summarize that:

- Default settings may not work great in most of the systems. Repeated autovacuum runs on a few tables while other tables starve for autovacuum is very common.
- Poor settings can result in autovacuum workers taking a considerable part of the server resources with little gain.
- Autovacuum has the natural tendency to start at the wrong time when a system undergoes heavy transactions.
- Practically, a scheduled vacuum job becomes necessary for those systems that undergo heavy transactions and, have a large number of transaction tables, and are expected to have spikes and peak time periods of load.

Clear analysis and tuning are important. It is always highly recommended to have a custom vacuum job that takes up your knowledge about the specific system and time windows of the least impact.

Overcoming VACUUM WRAPAROUND

By Robert Bernier

Robert is a Senior PostgreSQL Consultant at Percona, and whether it's tuning, high availability, high performance, data migration, trouble shooting, refactoring, or architecting a better mouse trap, he has a solution.

Transaction ID Wraparound occurs when the VACUUM process cannot keep up with database activity and the PostgreSQL service is forced to shut down.

In more technical parlance, Transaction ID Wraparound occurs when the semantics of Multi-Version Concurrency Control (MVCC) fail and when the number of unique transaction IDs reaches its maximum which numbers about two billion.

What leads up to this situation is when the VACUUM process managed by either the autovacuum workers or user-interaction (manual) does not keep up with the DML operations.

Transaction ID Wraparound can be caused by a combination of one or more of the following circumstances:

- Autovacuum is turned off
- Long-lived transactions
- Database logical dumps (on a REPLICA using streaming replication)
- Many session connections with locks extending across large swaths of the data cluster
- Intense DML operations forcing the cancellation of autovacuum worker processes

Transaction WRAPAROUND can cause a spontaneous shutdown of the Postgres database server in order to protect the integrity of the data.

PostgreSQL, at any one time, has a number of transactions that are tracked by a unique ID. Every so often, that number reaches the upper limit that can be registered, for example, 200 million transactions, which is the default and is then renumbered. But if the number of unique transaction IDs goes to its maximum transactions limit, known as TXID Wraparound, Postgres will force a shutdown in order to protect the data.

Here's how it works:

- Four billion transactions, 2^32, is the integer upper limit for the datatype used in Postgres.
- Two billion transactions, 2^31, is the upper limit that PostgreSQL permits before forcing a shutdown.
- Ten million transactions before the upper limit is reached, WARNING messages consisting of a countdown will be logged.
- One million transactions before the upper limit is reached, PostgreSQL goes to READ-ONLY mode.

Warning Signs

In the case of the autovacuum daemon falling behind across the entire data cluster, review your monitoring solution in order to identify the trend of these metrics:

- IO wait increases
- CPU load increases
- SQL performance decreases

Mitigation steps include:

- Reviewing the internal Postgres monitoring metrics and confirming tables are being vacuumed.
- Reviewing the Postgres logs, look for an overabundance of canceled autovacuum worker processes.
- Reviewing the view "pg_stat_activity" and looking for a query string – PREVENTING TRANSACTION ID WRAPAROUND -. Actually, this is a normal message. But one should not see autovacuum running solely for the purposes of mitigating WRAPAROUND.

Here are example error messages that you can find in the Postgres logs when threatened by a shutdown due to WRAPAROUND:

```
#
# When less than 10 million transactions remain before shutdown
#
WARNING: database "mydb" must be vacuumed within 177009986 transactions
HINT: To avoid a database shutdown, execute a database-wide VACUUM in "mydb".
```

Here's a set of queries that will help you determine if WRAPAROUND is a risk:

```
    --
    -- Database query for transaction age per database
    --  and as a percentage of maximum permitted transactions
    --
SELECT  datname,
       age(datfrozenxid),
       (age(datfrozenxid)::numeric/1000000000*100)::numeric(4,2) as "% WRAPAROUND RISK"
FROM pg_database ORDER BY 2 DESC;
--
-- Database query for transaction age per table
--
SELECT
c.oid::regclass as table_name,
greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as "TXID age",
(greatest(age(c.relfrozenxid),age(t.relfrozenxid))::numeric/1000000000*100)::numeric(4,2) as "%
WRAPAROUND RISK"
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm')
ORDER BY 2 DESC;
--
-- Other vacuum runtime parameters of interest
--  returning TXID age
--
SELECT  name, setting
FROM pg_settings
WHERE name ~ 'vacuum'
AND name ~'_age$'
ORDER BY 1 ASC;
```

Preventing Transaction ID Wraparound

First and foremost, make certain all tables are regularly vacuumed. A correctly configured autovacuum process takes care of this without it ever becoming an issue. Otherwise, you will need to consider a manual VACUUM strategy.

The following are merely suggestions since each situation is highly subjective.
If you have the time, run the following invocation of vacuumdb. The value for option '-j' can vary from a couple to a value equal to the number of CPUs on the host. The option '-a' will process each database in alphabetical order.

```
vacuumdb -F -a -z -j 10 -v
```

Consider a bash script targeting individual databases if you see one is more urgent than another:

```
vacuumdb -z -j 10 -v <mydatabase>
```

Immediate Action: Approaching Wraparound at < 10 Million Transactions

The following is the set of actions to take when TRANSACTION WRAPAROUND is imminent. Remember, you are in a race against time.

You must vacuum the entire data cluster before the remaining available transaction id drops to one million transactions.

Action

- The task is to vacuum the databases as quickly as possible.
- The tool of choice is the CLI "vacuumdb".
- Use as many threads as reasonable.
- Run VACUUM in verbose mode and log the output.
- Monitor log output for anomalous messages, i.e., vacuum fails, etc.
- Run "vacuumdb" against individual databases and, if necessary, individual tables.
- Avoid using the option '-a'.

Scripts

Here's a pair of example scripts that you can use as a starting point when developing your own mitigation protocol.

Method

- Identify the database with the oldest TXID
- Generate a list of tables in order of the oldest TXID age to the youngest
- Feed this list of tables into a script that invokes vacuumdb and VACUUM one table per invocation

The secret sauce is xargs, which enables one to utilize as many CPUs as reasonably possible. The following pair of bash scripts invoke vacuumdb against a series of tables. Of course, there's more than one way to do this.

Script one generates a list of tables in a selected database and calls script two, which executes the VACUUM on each of those tables individually.

SCRIPT ONE (go1_highspeed_vacuum.sh)

```
#!/bin/bash
#
# INVOCATION
# EX: ./go1_highspeed_vacuum.sh
#
#########################################################
# EDIT AS REQUIRED
export CPU=4
export PAGER=less PGUSER=postgres PGPASSWORD=mypassword PGDATABASE=db01 PGOPTIONS='-c statement_
timeout=0'
#########################################################

SQL1="
with a as (select   c.oid::regclass as table_name,
                    greatest(age(c.relfrozenxid),age(t.relfrozenxid))
           from pg_class c
           left join pg_class t on c.reltoastrelid = t.oid
           where c.relkind in ('r', 'm')
            order by 2 desc)
select table_name from a
"

LIST="$(echo "$SQL1" | psql -t)"

# the 'P' sets the number of CPU to use simultaneously
xargs -t -n 1 -P $CPU ./go2_highspeed_vacuum.sh $PGDATABASE<<<$LIST

echo "$(date): DONE"
```

SCRIPT TWO (go2_highspeed_vacuum.sh)

```
#!/bin/bash


#########################################################
# EDIT AS REQUIRED
export PAGER=less PGUSER=postgres PGPASSWORD=mypassword PGOPTIONS='-c statement_timeout=0'
export DB=$1

#########################################################

vacuumdb --verbose ${DB} > ${DB}.log 2>&1
```

Tips

- Be prepared to execute vacuumdb against the databases in REVERSE alphabetical order to avoid clashing with the autovacuum worker processes, which vacuums in FORWARD alphabetical order.
- Query table "pg_stat_activity".
- Always monitor where the autovacuum processes are working.
- Avoid working on the same table that the autovacuum workers are currently processing.
- Use the autovacuum workers as an indicator of what databases remain to be processed.
- Kill active autovacuum workers when in conflict with a manual vacuum in order to speed things up.

Immediate action: When PostgreSQL service has shut down due to transaction wraparound
One recovers from a forced shutdown due to transaction ID wraparound by performing a cluster-wide

vacuum in single-user mode:

Log in to the host, and as a UNIX user, “Postgres” executes an invocation that is something similar:

```
# it is understood that environment
# variable PGDATA points to the data cluster
#
postgres --single -D $PGDATA postgres <<< 'vacuum analyze'
```

I would suggest scripting the vacuum process because you'll need to log in to each database to perform the VACUUM.

Generate and edit a list of all the databases:

```
    postgres --single -D $PGDATA postgres <<< 'select datname from pg_database' \
| grep '"' | cut -d '"' -f 2 > list_db
```

Here is an example using the aforementioned list “list_db”:

```
#
# it is understood the database list has

#  been edited before invoking this code snippet
#
for u in $(cat list_db)
do
    postgres --single -D $PGDATA $u <<< 'vacuum analyze'
done
```

TXID Wraparound is one of the scariest scenarios that can occur. Thankfully, this is an extremely rare incident and only occurs when systems are either under extremely heavy load or have been neglected.

Don't get caught!

Remember: the best DBA is the one that's never noticed.

Partitioning

PostgreSQL partitioning is a technique used to enhance the performance and manageability of large tables by splitting them into smaller, more manageable pieces called partitions. This approach allows for more efficient data access and manipulation, especially in scenarios involving large volumes of data. Through partitioning, databases can achieve optimized query performance, thanks to partition pruning, and facilitate easier data management activities like archival and purging by allowing these tasks to be executed on individual partitions.

In this chapter, our experts provide an array of strategies for employing PostgreSQL partitioning effectively. These discussions cover not only the default partitioning techniques but also provide practical insights into leveraging pg_partman to automate and simplify partition management tasks.

While not a comprehensive guide to partitioning, it's a good jumping-off point for those looking to enhance the scalability and efficiency of their PostgreSQL database systems.

PostgreSQL Partitioning Using Traditional Methods

By Neha Korukula

Neha has been a Jr. PostgreSQL DBA in Managed Services at Percona since 2022 with extensive experience on installation, user configuration, setting up high availability, performance tuning, and migrations with minimal /zero downtime.

Partitioning is the concept of splitting large tables logically into smaller pieces for better database performance.

Methods of built-in PostgreSQL partition techniques

- Range partitioning
- List partitioning
- Hash partitioning

When to use partitioning

- Bulk operations like data loads and deletes can be performed using the partition feature of ATTACH and DETACH effectively.
- The exact point at which a table will benefit from partitioning depends on the application. However, a rule of thumb is that the size of the table should exceed the physical memory of the database server.
- As data grows, sub-partitions can be created, which enhances the performance, and old partitions can be deleted either by making them standalone or dropping them entirely.

Benefits of partitioning

- Query performance for DDL and DML operations can be improved in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions, which is explained below.
- When queries or updates access a large percentage of a single partition, performance can be improved by using a sequential scan of that partition instead of using an index, which would require random-access reads scattered across the whole table.
- Dropping the partition table or truncating the partition table can be done using DROP TABLE and TRUNCATE TABLE, respectively, reducing the load through DELETE operations.

Range partitioning

Database partition that is based on a specific range of columns with data like dates and Numeric values.

Here, as an example, I created a table with range partitioning and partition tables for each quarter on a Date column.

```
CREATE TABLE employees (id INT NOt NULL , fname VARCHAR(20) , lname VARCHAR (20) , dob DATE NOT NULL ,
joined DATE NOT NULL) PARTITION BY RANGE (joined);

CREATE TABLE employees_q1 PARTITION of employees for VALUES FROM ('2022-01-01') to ('2022-04-01');
CREATE TABLE employees_q2 PARTITION of employees for VALUES FROM ('2022-04-01') to ('2022-07-01');
CREATE TABLE employees_q3 PARTITION of employees for VALUES FROM ('2022-07-01') to ('2022-10-01');
CREATE TABLE employees_q4 PARTITION of employees for VALUES FROM ('2022-10-01') to ('2023-01-01');
```

Range partitions are seen below in the table structure.

```
d+ employees
                                        Partitioned table "public.employees"
Column | Type | Collation | Nullable | Default | Storage | Compression | Stats target | Description
-------+----------------------+-----------+----------+---------+----------+-------------+--------------+-----
id     | integer              |           | not null | | plain    | | |
fname  | character varying(20) |           |          | | extended | | |
lname  | character varying(20) |           |          | | extended | | |
dob    | date                 |           | not null | | plain    | | |
joined | date                 |           | not null | | plain    | | |

Partition key: RANGE (joined)
    Partitions: employees_q1 FOR VALUES FROM ('2022-01-01') TO ('2022-04-01'),
                employees_q2 FOR VALUES FROM ('2022-04-01') TO ('2022-07-01'),
                employees_q3 FOR VALUES FROM ('2022-07-01') TO ('2022-10-01'),
                employees_q4 FOR VALUES FROM ('2022-10-01') TO ('2023-01-01')
```

Inserted some random data for entries with 365 days a year.

```
INSERT INTO employees (id ,fname,lname,dob ,joined) VALUES ( generate_series(1, 365) ,(array[‘Os-
wald’, ‘Henry’, ‘Bob’, ‘Vennie’])[floor(random() * 4 + 1)],
(array[‘Leo’, ‘Jack’, ‘Den’, ‘Daisy’ ,’Woody’])[floor(random() * 5 + 1)], ‘1995-01-01’::date +
trunc(random() * 366 * 3)::int,
generate_series(‘1/1/2022’::date, ‘12/31/2022’::date, ‘1 day’));
```

Range partitioned data is seen as below distributed among its partitions.

```
SELECT employees_q1 , employees_q2 , employees_q3 , employees_q4 , employees_totalcnt from
( SELECT COUNT(*) FROM employees_q1 ) AS employees_q1,
( SELECT COUNT(*) FROM employees_q2 ) AS employees_q2,
( SELECT COUNT(*) FROM employees_q3 ) AS employees_q3,
( SELECT COUNT(*) FROM employees_q3 ) AS employees_q4 ,
( SELECT COUNT(*) FROM employees ) AS employees_totalcnt ;
employees_q1 | employees_q2 | employees_q3 | employees_q4 | employees_totalcnt
--------------+--------------+--------------+--------------+--------------------
(90)          | (91)          | (92)          | (92)          | (365)
(1 row)
```

Performance of DDL operations

Here, I created a table without a partition and inserted the same data, similar to the partitioned table.

A query plan is seen better for DDL operations when performed on data with a single partition or fewer partitions.

```
 CREATE TABLE employees_nopartition (id INT NOt NULL , fname VARCHAR(20) , lname VARCHAR (20) , dob
DATE NOT NULL , joined DATE NOT NULL) ;

INSERT INTO employees_nopartition (id ,fname,lname,dob ,joined) VALUES ( generate_series(1, 365)
,(array[‘Oswald’, ‘Henry’, ‘Bob’, ‘Vennie’])[floor(random() * 4 + 1)],
(array[‘Leo’, ‘Jack’, ‘Den’, ‘Daisy’ ,’Woody’])[floor(random() * 5 + 1)], ‘1995-01-01’::date +
trunc(random() * 366 * 3)::int,
generate_series(‘1/1/2022’::date, ‘12/31/2022’::date, ‘1 day’));

EXPLAIN select * from employees_nopartition where joined &gt;= ‘2022-05-12’ and joined &lt; ‘2022-
06-10’;
                               QUERY PLAN
------------------------------------------------------------------------
Seq Scan on employees_nopartition (cost=0.00..8.47 rows=29 width=22)
         Filter: ((joined &gt;= ‘2022-05-12’::date) AND (joined &lt; ‘2022-06-10’::date))
(2 rows)
```

Here we can see a better query plan when data is fetched from the partitioned table than data fetched from the non-partitioned table.

```
EXPLAIN select * from employees where joined &gt;= ‘2022-05-12’ and joined &lt; ‘2022-06-10’;
                               QUERY PLAN
------------------------------------------------------------------------
Seq Scan on employees_q2 (cost=0.00..2.37 rows=29 width=22)
    Filter: ((joined &gt;= ‘2022-05-12’::date) AND (joined &lt; ‘2022-06-10’::date))
(2 rows)
```

List partitioning

Database partition that is based on key value(s) or discrete values and partition can also be done with the expression of the column like (RANGE BY LIST(expression)), which is explained below.

For example, I created a table with a list partition and a few list-partitioned tables and inserted some random data with 1,000 rows.

```
 CREATE TABLE sales (id INT NOT NULL , branch VARCHAR(3),type text, Amount int ) PARTITION BY LIST
(branch);

CREATE TABLE HYD_sales PARTITION of sales for VALUES IN ('HYD');
CREATE TABLE BLR_sales PARTITION of sales for VALUES IN ('BLR');
CREATE TABLE DEL_sales PARTITION of sales for VALUES IN ('DEL');
CREATE TABLE TPT_sales PARTITION of sales for VALUES IN ('TPT');

INSERT into sales (id , branch ,type , amount ) VALUES ( generate_series(1, 1000) , (array['HYD',
'BLR', 'DEL', 'TPT'])[floor(random() * 4 + 1)] ,
(array['Laptops', 'Printers', 'Hardisks', 'Desktops' ,'Monitors'])[floor(random() * 5 + 1)], (ran-
dom()*200000)::int );
```

List partitions are seen in the table definition below:

```
  d+ sales
                                    Partitioned table "public.sales"
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description
--------+----------------------+-----------+----------+---------+----------+--------------+--------
-----
id     | integer              |   | not null |         | plain    | |
branch | character varying(3) |   |          |         | extended | |
type   | text                 |   |          |         | extended | |
amount | integer              |   |          |         | plain    | |
Partition key: LIST (branch)
       Partitions: blr_sales FOR VALUES IN ('BLR'),
                   del_sales FOR VALUES IN ('DEL'),
                   hyd_sales FOR VALUES IN ('HYD'),
                   tpt_sales FOR VALUES IN ('TPT')
```

Partitioned data distributed among its partitions is seen below:

```
SELECT blr_sales , del_sales , hyd_sales,tpt_sales, total_cnt from
( SELECT COUNT(*) FROM blr_sales ) AS blr_sales, ( SELECT COUNT(*) FROM del_sales ) AS del_sales,
( SELECT COUNT(*) FROM hyd_sales ) AS hyd_sales, ( SELECT COUNT(*) FROM tpt_sales ) AS tpt_sales ,
( SELECT COUNT(*) FROM sales ) AS total_cnt;
blr_sales | del_sales | hyd_sales | tpt_sales | total_cnt
-----------+-----------+-----------+-----------+-----------
(262)     | (258)     | (228)     | (252)     | (1001)
(1 row)
```

List partitioning using expression

For example, I created a table with list partitioning using the expression of a column.

```
CREATE TABLE donors (id INT NOt NULL , name VARCHAR(20) , bloodgroup VARCHAR (15) , last_donated
DATE ,
contact_num VARCHAR(10)) PARTITION BY LIST (left(upper(bloodgroup),3));

CREATE TABLE A_positive PARTITION of donors for VALUES IN ('A+ ');
CREATE TABLE A_negative PARTITION of donors for VALUES IN ('A- ');
CREATE TABLE B_positive PARTITION of donors for VALUES IN ('B+ ');
CREATE TABLE B_negative PARTITION of donors for VALUES IN ('B- ');
CREATE TABLE AB_positive PARTITION of donors for VALUES IN ('AB+');
CREATE TABLE AB_negative PARTITION of donors for VALUES IN ('AB-');
CREATE TABLE O_positive PARTITION of donors for VALUES IN ('O+ ');
CREATE TABLE O_negative PARTITION of donors for VALUES IN ('O- ');
```

List partitions are seen in the table definition below:

```
d+ donors
                                        Partitioned table "public.donors"
Column | Type | Collation | Nullable | Default | Storage | Compression | Stats target | Description
--------------+-----------------------+-----------+----------+---------+----------+-------------+-
-------------+-------------
id           | integer               | | not null | | plain | | |
name         | character varying(20) | |          | | extended | | |
bloodgroup   | character varying(15) | |          | | extended | | |
last_donated | date                  | |          | | plain | | |
contact_num  | character varying(10) | |          | | extended | | |
Partition key: LIST ("left"(upper((bloodgroup)::text), 3))
          Partitions: a_negative FOR VALUES IN ('A- '),
                      a_positive FOR VALUES IN ('A+ '),
                      ab_negative FOR VALUES IN ('AB-'),
                      ab_positive FOR VALUES IN ('AB+'),
                      b_negative FOR VALUES IN ('B- '),
                      b_positive FOR VALUES IN ('B+ '),
                      o_negative FOR VALUES IN ('O- '),
                      o_positive FOR VALUES IN ('O+ ')
```

Here, I inserted 100 random rows.

```
INSERT INTO donors (id , name , bloodgroup , last_donated , contact_num) VALUES (generate_series(1,
100) ,'user_' || trunc(random()*100) ,
(array['A+ group', 'A- group', 'O- group', 'O+ group','AB+ group','AB- group','B+ group','B-
group'])[floor(random() * 8 + 1)] , '2022-01-01'::date + trunc(random() * 366 * 1)::int,
CAST(1000000000 + floor(random() * 9000000000) AS bigint));
```

List partitioned data with expression distributed among its partitions is seen below:

```
SELECT a_negative , a_positive , ab_negative , ab_positive , b_negative ,b_positive ,o_negative , o_positive , total_
cnt from
( SELECT COUNT(*) FROM a_negative ) AS a_negative, ( SELECT COUNT(*) FROM a_positive ) AS a_positive,


( SELECT COUNT(*) FROM ab_negative ) AS ab_negative, ( SELECT COUNT(*) FROM ab_positive ) AS ab_positive ,
( SELECT COUNT(*) FROM b_negative ) AS b_negative, ( SELECT COUNT(*) FROM b_positive ) AS b_positive ,
( SELECT COUNT(*) FROM o_positive ) AS o_positive , ( SELECT COUNT(*) FROM o_negative ) AS o_negative,
( SELECT COUNT(*) FROM donors ) AS total_cnt;
 a_negative | a_positive | ab_negative | ab_positive | b_negative | b_positive | o_negative | o_positive | total_cnt
------------+------------+-------------+-------------+------------+------------+------------+------------+-----------
(9)         | (19)       | (10)        | (12)        | (12)       | (10)       | (18)       | (10)       | (100)
(1 row)
```

Performance of DML operations

Here is an example shown with the table, which is created without partitions and inserted the same data similar to that of the partitioned table.

Below, I created a table without a partition and inserted some random data with 1,000 rows to show query performance.

```
CREATE TABLE sales_nopartition (id INT NOT NULL , branch VARCHAR(3),type text, Amount int );

INSERT into sales_nopartition (id , branch ,type , amount ) VALUES ( generate_series(1, 1000) ,
(array[‘HYD’, ‘BLR’, ‘DEL’, ‘TPT’])[floor(random() * 4 + 1)] ,
(array[‘Laptops’, ‘Printers’, ‘Hardisks’, ‘Desktops’ ,’Monitors’])[floor(random() * 5 + 1)], (ran-
dom()*200000)::int );
```

UPDATE query performance

```
EXPLAIN update sales_nopartition set type = ‘Smart Watches’ where branch = ‘HYD’;
                          QUERY PLAN
-------------------------------------------------------------------------------
Update on sales_nopartition (cost=0.00..19.50 rows=229 width=50)
       -&gt; Seq Scan on sales_nopartition (cost=0.00..19.50 rows=229 width=50)
          Filter: ((branch)::text = ‘HYD’::text)
(3 rows)


EXPLAIN update sales set type = ‘Smart Watches’ where branch = ‘HYD’;
                           QUERY PLAN
-----------------------------------------------------------------
Update on sales (cost=0.00..5.10 rows=248 width=50)
Update on hyd_sales
       -&gt; Seq Scan on hyd_sales (cost=0.00..5.10 rows=248 width=50)
          Filter: ((branch)::text = ‘HYD’::text)
(4 rows)
```

DELETE query performance

```
EXPLAIN DELETE from sales_nopartition where branch='HYD';
                          QUERY PLAN
-----------------------------------------------------------------------
Delete on sales_nopartition (cost=0.00..19.50 rows=229 width=6)
       -&gt; Seq Scan on sales_nopartition (cost=0.00..19.50 rows=229 width=6)
          Filter: ((branch)::text = 'HYD'::text)
(3 rows)


EXPLAIN DELETE from sales where branch='HYD';
                         QUERY PLAN
-------------------------------------------------------------
Delete on sales (cost=0.00..5.10 rows=248 width=6)
Delete on hyd_sales
      -&gt; Seq Scan on hyd_sales (cost=0.00..5.10 rows=248 width=6)
         Filter: ((branch)::text = 'HYD'::text)
(4 rows)
```

The above examples show the performance of DELETE and UPDATE operations with data fetched from a single partitioned table having a better query plan than the one with no partitions.

Hash partitioning

Hash partitioning table is defined as the table partitioned by specifying a modulus and a remainder for each partition.

- Each partition will hold the rows for which the hash value of the partition key divided by the specified modulus will produce the specified remainder.
- Hash partitioning is best used when each partition is on different table spaces residing on separate physical disks, so the IO is equally divided by more devices.

For example, I created a table with hash partitioning and a few partitioned tables with modulus five.

```
CREATE TABLE students ( id int NOT NULL, name varchar(30) NOT NULL , course varchar(100) ,joined
date ) PARTITION BY hash(id);

CREATE TABLE student_0 PARTITION OF students FOR VALUES WITH (MODULUS 5,REMAINDER 0);
CREATE TABLE student_1 PARTITION OF students FOR VALUES WITH (MODULUS 5,REMAINDER 1);
CREATE TABLE student_2 PARTITION OF students FOR VALUES WITH (MODULUS 5,REMAINDER 2);
CREATE TABLE student_3 PARTITION OF students FOR VALUES WITH (MODULUS 5,REMAINDER 3);
CREATE TABLE student_4 PARTITION OF students FOR VALUES WITH (MODULUS 5,REMAINDER 4);
```

The table structure looks like the one below with five created partitions:

```
d+ students
                                Partitioned table “public.students”
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description
--------+------------------------+-----------+----------+---------+----------+--------------+-------------
id     | integer                | | not null | | plain | |
name   | character varying(30)  | | not null | | extended | |
course | character varying(100) | |          | | extended | |
joined | date                   | |          | | plain | |
Partition key: HASH (id)
        Partitions: student_0 FOR VALUES WITH (modulus 5, remainder 0),
                    student_1 FOR VALUES WITH (modulus 5, remainder 1),
                    student_2 FOR VALUES WITH (modulus 5, remainder 2),
                    student_3 FOR VALUES WITH (modulus 5, remainder 3),
                    student_4 FOR VALUES WITH (modulus 5, remainder 4)
```

Here, I Inserted some random data with 100,000 rows.

```
INSERT into students (id , name , course ,joined ) VALUES (generate_series(1, 100000) , ‘student_’
|| trunc(random()*1000) ,
(array[‘Finance &amp;amp; Accounts’, ‘Business Statistics’, ‘Environmental Science’])[floor(random() * 3
+ 1)],’2019-01-01’::date + trunc(random() * 366 * 3)::int);
```

We see below the hash partitioned data among its partitioned tables.

```
SELECT relname,reltuples as rows FROM pg_class WHERE relname IN (‘student_0’,’student_1’,’stu-
dent_2’,’student_3’,’student_4’) ORDER BY relname;
relname | rows
-----------+-------
student_0 | 19851
student_1 | 20223
student_2 | 19969
student_3 | 19952
student_4 | 20005
(5 rows)
```

Benefits of hash partitioning

- The primary benefit is to ensure an even distribution of data among a predetermined number of partitions.
- Hash keys are used effectively and efficiently in cases where ranges are not applicable, like employee number, product ID, etc.

What if the data is out of range or list?

For this purpose, we use default partitions on range and list partitioned tables.

For both range and list partitions, data can be stored temporarily, which is out-of-range, by creating a default partition and later creating an appropriate partition.

Hash-partitioned tables may not have a default partition, as the creation of a default partition for hash partitioning does not make any sense and is not needed.

Here, we see what happens when I try to insert data for which a partition doesn’t exist and how the default partition helps in this case.

```
INSERT into sales VALUES ( 1001 , 'MYS' , 'Scanners' , 190000);
ERROR: no partition of relation "sales" found for row
DETAIL: Partition key of the failing row contains (branch) = (MYS).

CREATE TABLE sales_default PARTITION of sales DEFAULT;
CREATE TABLE

INSERT into sales VALUES ( 1001 , 'MYS' , 'Scanners' , 190000);
INSERT 0 1

select * from sales_default ;
id | branch | type | amount
------+--------+----------+--------
1001 | MYS | Scanners | 190000
(1 row)
```

So the data we inserted is sent to the default partition, and partitions can be created later based on the data in the default table and available partitions.

Conclusion

Here, we discussed default partitioning techniques in PostgreSQL using single columns, and we can also create multi-column partitioning. PostgreSQL Partition Manager(pg_partman) can also be used for creating and managing partitions effectively.

Partitioning in PostgreSQL With pg_partman (Serial-Based & Trigger-Based)

By Neha Korukula

Neha has been a Jr. PostgreSQL DBA in Managed Services at Percona since 2022 with extensive experience on installation, user configuration, setting up high availability, performance tuning, and migrations with minimal /zero downtime.

The PostgreSQL partition manager pg_partman is an open source extension widely supported and actively maintained by the PostgreSQL community.

- pg_partman is an extension that streamlines the creation and management of table partition sets, supporting both time-based and serial-based partitioning approaches.
- You can use pg_partman to automate and create partitions by breaking large tables into smaller partitions, thereby enhancing performance.

Here, we will discuss the partitioning of newly created and already existing tables via serial-based.

In setting up partitioning without a template table using pg_partman, you actively configure the extension to create partitions based on specific criteria.

Using pg_partman, only the range partitioning (more about types of partition) can be implemented, either time-based or serial-based, but list partitioning cannot be implemented, as it can be only predicted when a new partition has to be created with range partitioning on dates or timestamps or IDs.

Partitioning by range without template

You can create partitions based on a serial-based column using pg_partman. For example:

Step one: Create a table with native partitioning type by range using serial-typed column.

```
partman=# CREATE TABLE students (id INT PRIMARY KEY , fname VARCHAR(20) , lname VARCHAR (20) , dob
DATE NOT NULL ,joined DATE )
PARTITION BY RANGE (id);
CREATE TABLE
```

Step two: Create parent to create initial child partitioned tables without template table.

```
partman=# SELECT partman.create_parent('public.students', p_control := 'id',p_type := 'native',p_
interval := '100000',p_premake := 3, p_start_partition := '0');
create_parent
---------------
t
(1 row)
```

The structure of the parent table with its partitions created is as follows.

```
partman=# d+ students
Partitioned table "public.students"
Column | Type                   | Collation | Nullable | Default | Storage | Compression | Stats
target | Description
--------+------------------------+-----------+----------+---------+----------+-------------+---
id     | integer                |           | not null |         | plain |
fname  | character varying(20) |           |          |         | extended |
lname  | character varying(20) |           |          |         | extended |
dob    | date                   |           | not null |         | plain |
joined | date                   |           |          |         | plain |
Partition key: RANGE (id)
Indexes:
        "students_pkey" PRIMARY KEY, btree (id)
Partitions: students_p100000 FOR VALUES FROM (100000) TO (200000),
            students_p200000 FOR VALUES FROM (200000) TO (300000),
            students_p300000 FOR VALUES FROM (300000) TO (400000),
            students_default DEFAULT
```

Here, we insert 1,000,000 rows to view data moving into partitioned tables.

```
partman=# INSERT INTO students (id ,fname,lname,dob ,joined) VALUES ( generate_series(1, 1000000)
,(array['Oswald', 'Henry', 'Bob', 'Vennie'])[floor(random() * 4 + 1)],
(array['Leo', 'Jack', 'Den', 'Daisy' ,'Woody'])[floor(random() * 5 + 1)], '1995-01-01'::date +
trunc(random() * 366 * 3)::int,
generate_series('1/1/2022'::date, '08/31/2023'::date, '1 day'));
INSERT 0 1000000

partman=# SELECT students_p100000 , students_p200000 ,students_p300000,
students_default,students_totalcnt from ( SELECT COUNT(*) FROM students_p100000 ) AS students_
p100000,
( SELECT COUNT(*) FROM students_p200000 ) AS students_p200000,( SELECT COUNT(*) FROM students_
p300000) AS students_p300000,
( SELECT COUNT(*) FROM students_default) AS students_default , ( SELECT COUNT(*) FROM students ) AS
students_totalcnt ;
students_p100000 | students_p200000 | students_p300000 | students_default | students_totalcnt
------------------+------------------+------------------+------------------+-------------------
(100000)         | (100000)         | (100000)         | (700000)         | (1000000)
(1 row)
```

Using functions, we can create child partitions and move the data from default to child tables.

```
partman.run_maintenance_proc - Create child partitions

partman=# CALL partman.run_maintenance_proc();
ERROR: updated partition constraint for default partition "students_default" would be violated by
some row
CONTEXT: SQL statement "ALTER TABLE public.students ATTACH PARTITION public.students_p400000 FOR
VALUES FROM ('400000') TO ('500000')"
```

Here, we see that, while performing run_maintenance_proc to create needed partitions for existing data in default tables, it's a violation of rows as it helps to create new partitions needed as per the data but not to insert data to new partitions. So, we can use the function partition_data_proc to move the data.

Using partman.partition_data_proc we can move data batch-wise to partitioned tables.

partman.partition_data_proc - Move data to partitioned tables

```
partman=# CALL partman.partition_data_proc ('public.students');'
NOTICE: Batch: 1, Rows moved: 99999 NOTICE: Batch: 2, Rows moved: 100000
NOTICE: Batch: 3, Rows moved: 100000 NOTICE: Batch: 4, Rows moved: 100000
NOTICE: Batch: 5, Rows moved: 100000 NOTICE: Batch: 6, Rows moved: 100000
NOTICE: Batch: 7, Rows moved: 100000 NOTICE: Batch: 8, Rows moved: 1
<strong>NOTICE: Total rows moved: 700000</strong>
NOTICE: Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning data
CALL
partman=# VACUUM ANALYZE public.students;
VACUUM
```

Here, we actively move the data from the default to the newly created partitioned table.

```
partman=# SELECT students_p100000 , students_p200000 ,students_p300000,
students_p400000 , students_p500000 ,students_p600000, students_p700000 , students_p800000 ,stu-
dents_p900000,students_default,
students_totalcnt from ( SELECT COUNT(*) FROM students_p100000 ) AS students_p100000, ( SELECT
COUNT(*) FROM students_p200000 )
AS students_p200000, ( SELECT COUNT(*) FROM students_p300000) AS students_p300000, ( SELECT
COUNT(*) FROM students_p400000 ) AS
students_p400000, ( SELECT COUNT(*) FROM students_p500000 ) AS students_p500000, ( SELECT COUNT(*)
FROM students_p600000)
AS students_p600000, ( SELECT COUNT(*) FROM students_p700000 ) AS students_p700000, ( SELECT
COUNT(*) FROM students_p800000 )
AS students_p800000, ( SELECT COUNT(*) FROM students_p900000) AS students_p900000, ( SELECT
COUNT(*) FROM students_default)
AS students_default , (SELECT COUNT(*) FROM students ) AS students_totalcnt ;
students_p100000 | students_p200000 | students_p300000 | students_p400000 | students_p500000 | stu-
dents_p600000 | students_p700000 |
students_p800000 | students_p900000 | <strong>students_default</strong> | students_totalcnt
------------------+------------------+------------------+------------------+------------------
(100000)         | (100000)         | (100000)         | (100000)         | (100000)         |
(100000)         | (100000)      |
(100000)         | (100000)         | <strong>(0)</strong>              | (1000000)
(1 row)
```

Partman type partitioning

Creating the parent table with the partman option (more about parent table creation options) allows us to create trigger-based partitions using pg_partman's method of partitioning, which has less read-write performance compared to native/declarative partitioning.

Below, let's see the implementation steps for trigger-based partitioning.

Partition creations using pg_partman for a non-declarative table (trigger-based)

Step one: Create a table with partman-type range partitioning using serial-typed columns and insert random data of 5,000 rows.

```
partman=# CREATE TABLE donors (id INT PRIMARY KEY , name VARCHAR(20) , bloodgroup VARCHAR (15) ,
last_donated DATE NOT NULL, contact_num VARCHAR(10));
CREATE TABLE

partman=# INSERT INTO donors (id , name , bloodgroup , last_donated , contact_num) VALUES (gener-
ate_series(1, 5000) ,'user_' || trunc(random()*100) ,
(array['A+ group', 'A- group', 'O- group', 'O+ group','AB+ group','AB- group','B+ group','B-
group'])[floor(random() * 8 + 1)] , '2022-01-01'::date + trunc(random() * 366 * 1)::int,
CAST(1000000000 + floor(random() * 9000000000) AS bigint));
INSERT 0 5000
```

Step two: Create parent to create initial child partitioned tables without template table.

```
partman=# SELECT partman.create_parent('public.donors', p_control := 'id',p_type := 'partman',p_in-
terval := '1000');
create_parent
---------------
t
(1 row)
```

The table structure has been created with child tables based on the default premake value of four on partman.part_config and a trigger named donors_part_trig.

```
partman=# d+ donors
Table “public.donors”
Column        | Type                  | Collation | Nullable | Default | Storage | Compression |
Stats target | Description
--------------+-----------------------+-----------+----------+---------+----------+-----------
id            | integer               |           | not null |         | plain |
name          | character varying(20) |           |          |         | extended |
bloodgroup    | character varying(15) |           |          |         | extended |
last_donated  | date                  |           | not null |         | plain |
contact_num   | character varying(10) |           |          |         | extended |
Indexes:
       “donors_pkey” PRIMARY KEY, btree (id)
Triggers:
donors_part_trig BEFORE INSERT ON donors FOR EACH ROW EXECUTE FUNCTION donors_part_trig_func()
Child tables: donors_p1000,donors_p2000,donors_p3000,donors_p4000, donors_p5000,
              donors_p6000,donors_p7000, donors_p8000,donors_p9000
Access method: heap
```

Using partman.partition_data_proc, we can move data batch-wise to partitioned tables.

```
partman=# CALL partman.partition_data_proc (‘public.donors’);
NOTICE: Batch: 1, Rows moved: 999 NOTICE: Batch: 2, Rows moved: 1000
NOTICE: Batch: 3, Rows moved: 1000 NOTICE: Batch: 4, Rows moved: 1000
NOTICE: Batch: 5, Rows moved: 1000 NOTICE: Batch: 6, Rows moved: 1
NOTICE: Total rows moved: 5000
NOTICE: Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning data
CALL
```

Consequently, upon inserting new data into the table, we notice that if the partition exists, the data seamlessly moves into the respective partitions; however, if the partition doesn’t exist, the data remains within the main table.

```
partman=# INSERT INTO donors (id , name , bloodgroup , last_donated , contact_num) VALUES (gener-
ate_series(5001, 7000) ,’user_’ || trunc(random()*100) ,
(array[‘A+ group’, ‘A- group’, ‘O- group’, ‘O+ group’,’AB+ group’,’AB- group’,’B+ group’,’B-
group’])[floor(random() * 8 + 1)] , ‘2022-01-01’::date + trunc(random() * 366 * 1)::int,
CAST(1000000000 + floor(random() * 9000000000) AS bigint));
INSERT 0 0
```

In this scenario, inserting data where the partition doesn't exist results in directing the data to the main table. Subsequently, manual movement of this data occurs in batches to the newly created partitions.

```
SELECT donors_p1000,donors_p2000,donors_p3000,donors_p4000,donors_p5000,
donors_p6000,donors_p7000,donors_p8000,donors_p9000,donors_totalcnt from
( SELECT COUNT(*) FROM donors_p1000 ) AS donors_p1000, ( SELECT COUNT(*) FROM donors_p2000) AS donors_p2000, (
SELECT COUNT(*) FROM donors_p3000 ) AS donors_p3000,
( SELECT COUNT(*) FROM donors_p4000 ) AS donors_p4000,( SELECT COUNT(*) FROM donors_p5000) AS donors_p5000,(
SELECT COUNT(*) FROM donors_p6000 ) AS donors_p6000,
( SELECT COUNT(*) FROM donors_p7000 ) AS donors_p7000,( SELECT COUNT(*) FROM donors_p8000) AS donors_p8000, (
SELECT COUNT(*) FROM donors_p9000) AS donors_p9000,
( SELECT COUNT(*) FROM donors ) AS donors_totalcnt ;
donors_p1000 | donors_p2000 | donors_p3000 | donors_p4000 | donors_p5000 | donors_p6000 | donors_p7000 | do-
nors_p8000 |
donors_p9000 | donors_totalcnt
--------------+--------------+--------------+--------------+--------------+--------------+--------------+------
--------+--------------+-----------------
(1000)        | (1000)       | (1000)       | (1000)       | (1000)       | (1000)       | (0)          | (0)
|
(0)           | (6000)
(1 row)

partman=# INSERT INTO donors (id , name , bloodgroup , last_donated , contact_num) VALUES (generate_se-
ries(14001, 15000) ,'user_' || trunc(random()*100) ,
(array['A+ group', 'A- group', 'O- group', 'O+ group','AB+ group','AB- group','B+ group','B- group'])[floor(ran-
dom() * 8 + 1)] , '2022-01-01'::date + trunc(random() * 366 * 1)::int,
CAST(1000000000 + floor(random() * 9000000000) AS bigint));
INSERT 0 1000
partman=# CALL partman.partition_data_proc('public.donors');
NOTICE: Batch: 1, Rows moved: 999
NOTICE: Batch: 2, Rows moved: 1
NOTICE: Total rows moved: 1000
NOTICE: Ensure to VACUUM ANALYZE the parent (and source table if used) after partitioning data
CALL
partman=# VACUUM ANALYZE donors;
VACUUM
```

Employing functions to proactively create partitioned tables and migrate data enables the proactive planning of partitions ahead of inserts, preventing data accumulation in default partitions or the main table.

Schema design

Schema design in PostgreSQL focuses on optimizing database architecture for efficient data storage, retrieval, and processing and involves setting up tables, defining relationships, choosing appropriate data types, and implementing constraints to maintain data integrity and enhance performance. Tailoring the design to the application's specific needs is crucial, and in this chapter, we highlight some of the approaches recommended by our experts to craft efficient and robust database schemas in PostgreSQL

Diffing PostgreSQL Schema Changes

By Robert Bernier

Robert is a Senior PostgreSQL Consultant at Percona, and whether it's tuning, high availability, high performance, data migration, trouble shooting, refactoring, or architecting a better mouse trap, he has a solution.

One of the routine operations when administering PostgreSQL is periodic updates to the database system's architecture. PostgreSQL does a good job of allowing one to update a schema, add types, functions, triggers, or alter a table by adding and removing columns and updating column data types, etc., in a reliable manner. However, there is no built-in mechanism to help identify the differences, let alone generate the necessary SQL, to accomplish updates in an easy manner from the development to the production environment.

So, let's talk about possible approaches to schema changes.

Using logical dump manifests

The easiest way to identify changes between schemas from one database to another is to compare schema dump manifests.

The following example demonstrates an approach one can take looking for differences between schema on different databases:

EXAMPLE:

```
-- create database schemas
create database db01
create database db01<br /><br /><br />

-- db01: version 1
create table t1 (
    c1 int,
    c2 text,
    c4 date
);

create table t2(
    c1 int,
    c2 varchar(3),
    c3 timestamp,
    c4 date
);

-- db02: version 2
create table t1 (
    c1 serial primary key,
    c2 varchar(256),
    c3 date default now()
);

create table t2(
    c1 serial primary key,
    c2 varchar(3),
    c3 varchar(50),
    c4 timestamp with time zone default now(),
    c5 int references t1(c1)
);

create index on t2 (c5);

# generate schema dumps
pg_dump -s db01 -Fc > db01.db
pg_dump -s db02 -Fc > db02.db

# generate manifests
pg_restore -l db01.db > db01_manifest.ini
pg_restore -l db02.db > db02_manifest.ini
```

This snippet demonstrates looking for differences by comparing the md5 checksums:

```
# EX 1: generate checksums
md5sum
<(tail +16 db01_manifest.ini | cut -d ' ' -f 4-)
<(tail +16 db02_manifest.ini | cut -d ' ' -f 4-)

# output
$ 9d76c028259f2d8bed966308c256943e  /dev/fd/63
$ ba124f9410ea623085c237dc4398388a  /dev/fd/62
```

This next snippet diffs the differences between the two manifests, identifying only those objects and attributes that have changed. Notice that redundant information, the first 16 lines, are skipped:

```
# EX 2: perform diff
diff
> <(tail +16 db01_manifest.ini | cut -d ' ' -f 4-)
> <(tail +16 db02_manifest.ini | cut -d ' ' -f 4-)
```

This resultant diff shows the changes made between the two schemas:

```
1a2,3
> SEQUENCE public t1_c1_seq postgres
> SEQUENCE OWNED BY public t1_c1_seq postgres
2a5,12
> SEQUENCE public t2_c1_seq postgres
> SEQUENCE OWNED BY public t2_c1_seq postgres
> DEFAULT public t1 c1 postgres
> DEFAULT public t2 c1 postgres
> CONSTRAINT public t1 t1_pkey postgres
> CONSTRAINT public t2 t2_pkey postgres
> INDEX public t2_c5_idx postgres
> FK CONSTRAINT public t2 t2_c5_fkey postgres
```

The good news is that there are a number of existing tools that can reconcile differences between a proposed schema design and the target schema:

- Commercial offerings can differentiate schema between databases in an elegant and efficient manner. Researching, ala Google, yields the most popular technologies one can use.
- In regards to open source solutions, there are a number of projects capable of diffing Postgres database schemas.

Working with the apgdiff extension

The following is an example implementation of the open source tool apgdiff.

Apgdiff can be found in the Postgres community repository. It compares two schema dump files and creates an SQL output file that is, for the most part, suitable for upgrades of old schemata:

```
Package: apgdiff

Version: 2.7.0-1.pgdg18.04+1
Architecture: all
Maintainer: Debian PostgreSQL Maintainers <team+postgresql@tracker.debian.org>
Installed-Size: 173
Depends: default-jre-headless | java2-runtime-headless
Homepage: https://www.apgdiff.com/
Priority: optional
Section: database
Filename: pool/main/a/apgdiff/apgdiff_2.7.0-1.pgdg18.04+1_all.deb
Size: 154800
SHA256: 9a83fcf54aed00e1a28c3d00eabe1c166977af1e26e91035e15f88b5215b181b
SHA1: ea713acb55898f07374dadd1bebb09ec2fa4b589
MD5sum: e70a97903cb23b8df8a887da4c54e945
```

The following example demonstrates how one can update differences between the development environment and the production database schema using apgdiff.

EXAMPLE:

```
apt install -y apgdiff


# EX 1: dump as SQL statements
pg_dump -s db01 -Fp > db01.sql
pg_dump -s db02 -Fp > db02.sql
createdb db03 --template=db01

apgdiff --ignore-start-with db01.sql db02.sql > db01-db02.sql

# “psql -1” encapsulates statements within a transaction
psql -1 -f db01-db02.sql db03

# EX 2: uses logical dumps
# notice the dumps are standard logical dumps and includes data
pg_dump db01 -Fc > db01.db
pg_dump db02 -Fc > db02.db
createdb db03 --template=db01


# this invocation assumes the resultant diff doesn’t require editing
apgdiff --ignore-start-with
    <(pg_restore -s -f - db01.db)
    <(pg_restore -s -f - db02.db)
    | psql -1 db03
```

There’s more you can accomplish with these simple approaches. By incorporating variations of these, one can create fairly sophisticated shell scripts with little code and, with a little luck, not that much effort.

Load balancing/ connection pooling

As your database expands, managing and optimizing its performance can become increasingly difficult. This is where tools like Pgpool-II, PgBouncer, and HAProxy can come into play. In this chapter, we highlight their use cases and benefits and demonstrate using a combination of PgBouncer and HAProxy in order to scale PostgreSQL.

These insights from our experts, while not encyclopedic on all things load balancing and connection pooling, are sure to help you enhance your PostgreSQL environment.

Pgpool-II Use Cases and Benefits

By Kai Wagner

Kai works as a Sr. Engineering Manager for PostgreSQL at Percona and is actively engaged and a public speaker for many open source projects such as ceph, openATTIC, Linux Kernel, and now PostgreSQL.

PostgreSQL is a popular open source relational database management system many organizations use to store and manage their data. One of the key benefits of using PostgreSQL is its reliability, scalability, and performance. However, as the size of your database grows, it can become challenging to manage and optimize its performance.

Pgpool-II

This is where the pgpool-II comes in. Pgpool-II is a powerful tool that can help you manage and optimize the performance of your PostgreSQL database cluster. Let's explore the benefits of using pgpool-II for PostgreSQL.

1. Load balancing

One of the primary benefits of using pgpool-II is its ability to distribute incoming client connections across multiple PostgreSQL servers, allowing you to balance the load and increase the capacity of your database cluster. This means that as your application grows and more clients connect to the database, pgpool-II can distribute the load across multiple servers, preventing any one server from becoming overwhelmed.

2. Connection pooling

Pgpool-II can maintain a pool of idle database connections that can be reused by multiple clients, reducing the overhead of establishing new connections and improving application performance. This is particularly useful for applications that frequently open and close database connections, as it reduces the time it takes to establish a connection.

3. Query caching

Pgpool-II can cache frequently used queries in memory, reducing the load on your PostgreSQL servers and improving response times. This means that when a query is executed, pgpool-II can check the cache first to see if the results are already available rather than sending the query to the database server. This can significantly improve query response times and reduce the load on your database servers.

4. High availability

Pgpool-II can automatically redirect client connections to a backup server when a PostgreSQL server becomes unavailable. Furthermore, the online recovery feature allows new standby nodes to be added seamlessly. Moreover, Pgpool's watchdog module ensures that if one node goes down, another node takes over its place, resulting in minimal to no interruption for client applications. With automatic failover, online recovery, and watchdog, your application can continue functioning without interruption or manual intervention, even if one or more database servers or pgpool-II servers go down.

5. Online recovery

Pgpool-II allows you to perform online recovery of a failed PostgreSQL server without interrupting service, minimizing downtime and ensuring data integrity. This means that if a server fails, you can perform recovery operations without interrupting the service, allowing your application to continue operating as normal.

In conclusion, the pgpool-II extension provides a range of powerful features that can help you optimize the performance, availability, and scalability of your PostgreSQL database cluster. Whether you need load balancing, connection pooling, query caching, high availability, parallel query execution, or online recovery, pgpool-II has you covered.

By using pgpool-II, you can improve the reliability and performance of your PostgreSQL database cluster, ensuring that your application continues to function smoothly as your database grows. Percona fully supports Pgpool-II; you can download the latest builds from our repository.

Scaling PostgreSQL using Connection Poolers and Load Balancers for an Enterprise Grade environment

By Avinash Vallarapu

Avinash joined Percona in May 2018 and was PostgreSQL Tech Lead - Global Services until 2021. Before joining Percona, Avi worked as a Database Architect at OpenSCG for two years and as a DBA Lead at Dell for 10 Years in Database technologies.

Jobin Augustine

Jobin is a PostgreSQL Escalation Specialist in Support at Percona and open source advocate with more than 21 years of experience as a consultant, architect, administrator, writer, and trainer in PostgreSQL, Oracle, and other database technologies. He has always participated actively in the open source communities, and his main focus area is database performance and optimization.

and Fernando Laudares Camargos

As our application grows, how do we accommodate an increase in traffic while maintaining the quality of the service (response time)? The answer to this question depends on the nature of the workload at play, but it is often shaped around:

(a) improving its efficiency and
(b) increasing the resources available

Why connection pooling?

When it comes to improving the efficiency of a database workload, one of the first places we start looking at is the list of slow queries; if the most popular ones can be optimized to run faster, then we can easily gain some overall performance back. Arguably, we may look next at the number and frequency of client connections: is the workload composed of a high number of very frequent but short-lived connections? Or are clients connections of a more moderate number and tend to stick around for longer?

If we consider the first scenario further–a high number of short-lived connections–and that each connection spawns a new OS process, the server may hit a practical limit as to the number of transactions—or connections—it can manage per second, considering the hardware available and the workload being processed. Remember that PostgreSQL is process-based, as opposed to thread-based, which is itself an expensive operation in terms of resources, both CPU and memory.

A possible remedy for this would be the use of a connection pooler, acting as a mediator between the application and the database. The connection pooler keeps a number of connections permanently opened with the database and receives and manages all incoming requests from clients itself, allowing them to temporarily use one of the connections it already has established with PostgreSQL. This removes the burden of creating a new process each time a client establishes a connection with PostgreSQL and allows it to employ the resources that it would otherwise use for this to serve more requests (or complete them faster).

Rule of thumb?

A general rule of thumb that we often hear is that you may need a connection pooler once you reach around 350 concurrent connections. However, the actual threshold is highly dependent on your database traffic and server configuration: as we found out recently, you may need one much sooner.

You may implement connection pooling using your native application connection pooler (if there is one available) or through an external connection pooler such as PgBouncer and pgPool-II. For the solution we have built, which we demonstrated in our webinar on October 10, we have used PgBouncer as our connection pooler.

PgBouncer

PgBouncer is a lightweight (thread-based) connection pooler that has been widely used in PostgreSQL based environments. It "understands" the PostgreSQL connection protocol and has been a stable project for over a decade.

PgBouncer allows you to configure the pool of connections to operate in three distinct modes: session, statement, and transaction. Unless you have a good reason to reserve a connection in the pool to a single user for the duration of its session or are operating with single-statements exclusively, transaction mode is the one you should investigate.

A feature that is central to our enterprise-grade solution is that you can add multiple connection strings using unique alias names (referred to as database names). This allows greater flexibility when mediating connections with multiple database servers. We can then have an alias named "master_db" that will route connections to a master/primary server and another alias named "slave_db" that will route connections to a slave/standby server.

Scaling up

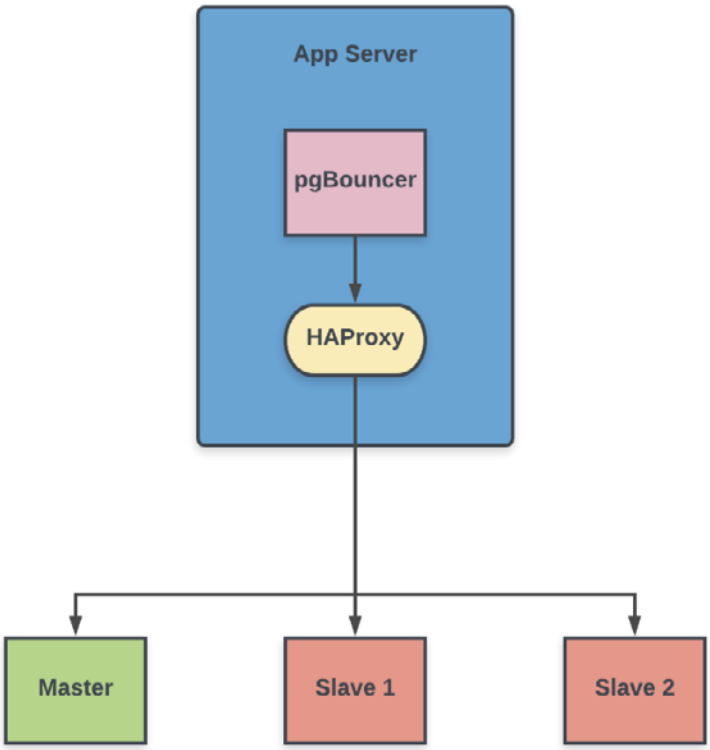
Once efficiency is taken care of, we can then start working on increasing the resources or computing power available to process database requests. Scaling vertically means, in short, upgrading the server to more and faster cores, memory, and storage. It's a simple approach but one that reaches a practical limitation rather quickly. It is not in line with other requirements of an enterprise-grade solution, such as high availability. The alternative is scaling horizontally. As briefly introduced above, a common way for implementing horizontal scalability is to redirect reads to standby servers (replicas) with the help of a proxy, which can also act as a load balancer, such as HAProxy. We'll be discussing these ideas further here and showcase their integration in our webinar.

HAProxy

HAProxy is a popular open source TCP/HTTP load balancer that can distribute the workload across multiple servers. It can be leveraged in a PostgreSQL replication cluster that has been built using streaming replication. When you build replication using the streaming replication method, standby replicas are open for reads. With the help of HAProxy, you can efficiently utilize the computing power of all database servers, distributing read requests among the available replicas using algorithms such as Least Connection and Round Robin.

A combination of connection pooler and load balancer to scale PostgreSQL

The following diagram represents a simplified part of the architecture that composes the enterprise-grade solution we've designed, where we employ PgBouncer and HAProxy to scale our PostgreSQL cluster:



Our PgBouncer contains two database (alias) names, one for redirecting writes to the master and another for balancing reads across standby replicas, as discussed above. Here is how the database section looks in the pgbouncer.ini, PgBouncer's main configuration file:

```
[databases]
master = host=haproxy port=5002 dbname=postgres
slave = host=haproxy port=5003 dbname=postgres
```

Notice that both database entries redirect their connections to the HAProxy server, but each to a different port. The HAProxy, in turn, is configured to route the connections in functions of the incoming port they reach. Considering the above pgBouncer config file as a reference, writes (master connections) are redirected to port 5002, and reads (slave connections) to port 5003. Here is how the HAProxy config file looks:

```
# Connections to port 5002
listen Master
    bind *:5002
    option tcp-check
    tcp-check send GET\ / HTTP/1.0\r\n
    tcp-check send HOST\r\n
    tcp-check send \r\n
    tcp-check expect string “role”:\ “master”
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server pg0 pg0:5432 maxconn 100 check port 8008
    server pg1 pg1:5432 maxconn 100 check port 8008
    server pg2 pg2:5432 maxconn 100 check port 8008


# Connections to port 5003
listen Slaves
    bind *:5003
    option tcp-check
    tcp-check send GET\ / HTTP/1.0\r\n
    tcp-check send HOST\r\n
    tcp-check send \r\n
    tcp-check expect string “role”:\ “replica”
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server pg0 pg0:5432 maxconn 100 check port 8008
    server pg1 pg1:5432 maxconn 100 check port 8008
    server pg2 pg2:5432 maxconn 100 check port 8008
```

As seen above:

- connections to port 5002 are sent to server with role: “master”
- connections to port 5003 are sent to server with role: “replica”

HAProxy relies on Patroni to determine the role of the PostgreSQL server. Patroni is being used here for cluster management and automatic failover. By using Patroni’s REST API (on port 8008 in this scenario), we can obtain the role of a given PostgreSQL server. The example below shows this in practice, the IP addresses denoting the PostgreSQL servers in this setup:

```
$ curl -s ‘http://192.168.50.10:8008’ | python -c “import sys, json; print json.load(sys.stdin)
[‘role’]”
replica
$ curl -s ‘http://192.168.50.20:8008’ | python -c “import sys, json; print json.load(sys.stdin)
[‘role’]”
master
$ curl -s ‘http://192.168.50.30:8008’ | python -c “import sys, json; print json.load(sys.stdin)
[‘role’]”
replica
```

HAProxy can thus rely on Patroni’s REST API to redirect connections from the master alias in PgBouncer to a server with role master. Similarly, HAProxy uses server role information to redirect connections from a slave alias to one of the servers with role replica using the appropriate load balancer algorithm.

This way, we ensure that the application uses the advantage of a connection pooler to leverage connections to the database and also of the load balancer, which distributes the read load to multiple database servers as configured.

There are many other open source connection poolers and load balancers available to build a similar setup. You can choose the one that best suits your environment—just make sure to test your custom solution appropriately before bringing it to production.

PostGIS

This chapter takes a look into PostGIS, a powerful open source tool that extends the capabilities of PostgreSQL by introducing support for storing, indexing, and querying geospatial data. Recommended by our expert, this section focuses on utilizing PostGIS to unlock superior performance within PostgreSQL environments.

While not meant to be an exhaustive manual on optimizing with PostGIS, it aims to provide you with the insights necessary to get started.

Using PostGIS To Enable Better Performance in PostgreSQL

By Robert Bernier

Robert is a Senior PostgreSQL Consultant at Percona, and whether it's tuning, high availability, high performance, data migration, trouble shooting, refactoring, or architecting a better mouse trap, he has a solution.

Mastering Geographical Information Systems, better known simply as GIS, can be considered in some ways as a rite of passage. The complexities and challenges involved in learning, which are ostensibly non-IT concepts, are steep. However, as they say, "There's more than one way to skin a cat." I'd like to share one way to tackle this challenge with you.

Let me introduce you to PostGIS.

What is PostGIS?

PostGIS is a PostgreSQL extension that adds GIS capabilities to this RDBMS. Its popularity stems not only from being "free" but also from the fact that it's considered to be among the leading GIS implementations in the world today. Virtually every major front-end application provides the hooks for a PostGIS, PostgreSQL-enabled back-end.

The PostGIS project, which is BSD-licensed, began back in 2001. It turns our vanilla-flavored Postgres into a spatial database and includes spatial datatypes (geometry, geography), spatial indexes (r-tree, quad-tree, kd-tree), and spatial functions.

Essential PostGIS Tools and Applications

Working with GIS normally requires several layers of technology of Geo-Spatial Software, for example:

Boundless Server (formerly the OpenGeo Suite)

- PostGIS – Spatially enabled object-relational database.
- GeoServer – Software server for loading and sharing geospatial data
- GeoWebCache – Tile cache server that accelerates the serving of maps
- Composer – Web-based map configuration and styling utility
- WPS Builder – Web-based graphical constructor for generating server-side processes
- QuickView – Web application for composing and styling web maps

Note: Boundless Server GitHub repository

Advantages of Using PostGIS for Better Performance

PostGIS offers a solution to provide geospatial data capabilities in PostgreSQL for indexing, storing, and querying geographical data.

Spatial Indexing

Spatial indexing accelerates the querying and retrieval of geospatial data in PostgreSQL, improving response times for applications and analytics. It uses various spatial indexing techniques, such as R-tree and GiST (Generalized Search Tree), to organize data based on their spatial relationships, allowing the database to quickly narrow down the search space.

Geospatial Functions and Queries

PostGIS offers a wide array of geospatial functions for PostgreSQL, allowing users to perform complex spatial operations such as distance and area computations, geometric alterations, and buffering. Optimized spatial queries, like finding nearby points or intersecting polygons, use these functions to retrieve geospatial data efficiently.

Spatial Data Types

PostGIS supports spatial data types like Point, LineString, and Polygon in 2D and 3D. These allow efficient spatial data storage and retrieval due to their optimized indexing mechanisms. They enable spatial indexing, enabling faster query processing by organizing data based on geometric properties.

Querying The Spatial Database

It's amazing the kinds of answers one can get by a single, well-composed, and yet standard query by asking such simple questions as:

- How far is it from here to there?
- What's the closest point between two meandering streets?
- Is a street found in a certain zip code?
- How many homes are susceptible to flooding?

Here are a few example queries using a PostGIS-powered database. Refer here for more community-related shapefiles for Kamloops, British Columbia:

```
--
-- EXAMPLE 1: What is the total length of all roads in British Columbia?
--
SELECT sum(ST_Length(the_geom))/1000 AS km_roads FROM bc_roads;

km_roads
------------------
70842.1243039643

--
-- EXAMPLE 2: How large is the city of Prince George, British Columbia?
--
SELECT ST_Area(the_geom)/10000 AS hectares
FROM bc_municipality
WHERE name = 'PRINCE GEORGE';

hectares
------------------
32657.9103824927

--
-- EXAMPLE 3: What is the largest municipality in the province of British Columbia?
--
SELECT name,
       ST_Area(the_geom)/10000 AS hectares
FROM bc_municipality
ORDER BY hectares DESC
LIMIT 1;

          name | hectares
---------------+-----------------
 TUMBLER RIDGE | 155020.025561

--
-- EXAMPLE 4: What is the length of roads fully contained within each municipality?
--
SELECT m.name,
       sum(ST_Length(r.the_geom))/1000 as roads_km
FROM   bc_roads AS r,
       bc_municipality AS m
WHERE ST_Contains(m.the_geom,r.the_geom)
GROUP BY m.name
ORDER BY roads_km;

                       name | roads_km
----------------------------+------------------
                     SURREY | 1539.47553551242
                  VANCOUVER | 1450.33093486576
            LANGLEY DISTRICT | 833.793392535662
                    BURNABY | 773.769091404338
               PRINCE GEORGE | 694.375543691
```

PostGIS Use Cases

PostGIS extends PostgreSQL's capabilities to handle geospatial data for solutions in real-world scenarios. From mapping to visualizations to logistics, PostGIS helps users to make informed decisions based on spatial insights.

Location-Based Services

PostGIS plays a role in location-based applications by enabling spatial data storage and analysis. For instance, a grocery delivery service could utilize PostGIS to optimize delivery for certain areas. A query that finds available drivers within a certain distance, facilitated by PostGIS's spatial indexing, would significantly enhance the platform's responsiveness.

Geospatial Analytics

PostGIS utilizes geospatial data analysis by offering advanced tools for operations such as identifying the nearest points or calculating areas within polygons. The functions are optimized through spatial indexing for quicker and better queries of complex geospatial analyses, improving overall performance.

Mapping and Visualization

PostGIS improves map rendering and visualization through efficient geospatial data handling. It allows for quickly retrieving map data to ensure smooth and responsive displays.

How to Install the PostGIS Extension for PostgreSQL

Basic Installation

The following installation instructions assume one is using PostgreSQL version 12 on Linux/CENTOS-7, although any major version of Postgres and OS can be used.

```
#
# REDHAT/CENTOS DERIVATIVES, USING COMMUNITY POSTGRES
#
yum update -y
yum install -y postgis30_12 postgis30_12-client

#
# CREATE THE DATACLUSTER
#
/usr/pgsql-12/bin/postgresql-12-setup initdb


#
# SERVICE STARTUP
#
systemctl start postgresql-12


#
# INSTALLING THE POSTGIS EXTENSION
#   Execute as UNIX account postgres
#
createdb mydatabase;
psql -c 'create extension postgis' mydatabase
```

Complete Installation for PostGIS Extension

The following provides a more complete installation of all PostGIS capabilities. Refer to the PostGIS documentation for more information.

```
--
-- if you want to install raster support
--
CREATE EXTENSION postgis_raster;

--
-- if you want to install topology support
--
CREATE EXTENSION postgis_topology;

--
-- if you want to install sfcgal support
--
CREATE EXTENSION postgis_sfcgal;

--
-- if you want to install tiger geocoder
--
CREATE EXTENSION fuzzystrmatch
CREATE EXTENSION postgis_tiger_geocoder;

--
-- if you want to install pcre, add the address standardizer extension
--
CREATE EXTENSION address_standardizer;

--
-- list of all installed extensions
--
mydatabase=# select * from pg_available_extensions where name ~ '^postgis';
                     name | default_version | installed_version | comment
--------------------------+-----------------+-------------------+---------------------------------
-----------------------------------
                postgis-3 | 3.0.1           |                   | PostGIS geometry, geography, and
raster spatial types and functions
         postgis_raster-3 | 3.0.1           |                   | PostGIS raster types and
functions
         postgis_sfcgal-3 | 3.0.1           |                   | PostGIS SFCGAL functions
postgis_tiger_geocoder-3  | 3.0.1           |                   | PostGIS tiger geocoder and
reverse geocoder
       postgis_topology-3 | 3.0.1           |                   | PostGIS topology spatial types
and functions
                  postgis | 3.0.1           | 3.0.1             | PostGIS geometry, geography, and
raster spatial types and functions
           postgis_raster | 3.0.1           | 3.0.1             | PostGIS raster types and
functions
           postgis_sfcgal | 3.0.1           | 3.0.1             | PostGIS SFCGAL functions
   postgis_tiger_geocoder | 3.0.1           | 3.0.1             | PostGIS tiger geocoder and
reverse geocoder
         postgis_topology | 3.0.1           |                   | PostGIS topology spatial types
and functions
```

Working With Shapefiles

There are literally hundreds of terabytes available online. Ironically, the most precious data one can get is free to download because it's been generated by governments from all over the world for the public good.

Now, it's time to get some data. The most common format is "shapefile."

PostGIS includes these two command-line utilities:

- pgsql2shp
- shp2pgsql

Working with zip codes from the US Census data: TIGER files

TIGER/Line files are a digital database of geographic features, such as roads, railroads, rivers, lakes, legal boundaries, census statistical boundaries, etc., covering the entire United States and are freely available here.

Once you've navigated to this website, you can download a multitude of fascinating pieces of data.

Example:

Step one: Execute the following as a bash script

```
psql -U postgres &lt;&lt;_eof_
set ON_ERROR_STOP on

drop database if exists gis_demo;
create database gis_demo;

c gis_demo

create extension if not exists postgis;
create extension if not exists postgis_topology;

create schema gis;

create or replace function gis.findzipcode(
inout longitude double precision,
inout latitude double precision,
out zipcode varchar
) as
$$
begin
select zcta5ce10
into zipcode
from zipcodes, (values (st_makepoint(longitude,latitude)))t(p)
where st_contains(geom, p);
end;
$$
language plpgsql
security definer
set search_path=gis,public,topology;
_eof_
```

Step two: Download the 2019 zip code shapefile

The shapefile is stored in a zip file. Unzip the entire archive before attempting an upload.
URL: https://www.census.gov/cgi-bin/geo/shapefiles/index.php

```
#
# Execute the following, the actual arguments will vary
#
shp2pgsql -c -D -I tl_2019_us_zcta510.shp gis.zipcodes | psql -U postgres gis_demo
```

Once the upload into the Postgres database is complete, this is what you'll get:

```
gis_demo=# d
List of relations
   Schema | Name              | Type     | Owner
----------+-------------------+----------+----------
   public | geography_columns | view     | postgres
   public | geometry_columns  | view     | postgres
   public | spatial_ref_sys   | table    | postgres
 topology | layer             | table    | postgres
 topology | topology          | table    | postgres
 topology | topology_id_seq   | sequence | postgres
(6 rows)

gis_demo=# dn
List of schemas
     Name | Owner
----------+----------
      gis | postgres
   public | postgres
 topology | postgres
(3 rows)

gis_demo=# d zipcodes
Table "gis.zipcodes"
     Column | Type                    | Collation | Nullable | Default
------------+-------------------------+-----------+----------+-------------------------------
----
        gid | integer                 |           | not null | nextval('zipcodes_gid_
seq'::regclass)
  zcta5ce10 | character varying(5)    |           |          |
    geoid10 | character varying(5)    |           |          |
  classfp10 | character varying(2)    |           |          |
    mtfcc10 | character varying(5)    |           |          |
 funcstat10 | character varying(1)    |           |          |
    aland10 | double precision        |           |          |
   awater10 | double precision        |           |          |
 intptlat10 | character varying(11)   |           |          |
 intptlon10 | character varying(12)   |           |          |
       geom | geometry(MultiPolygon)  |           |          |
Indexes:
"zipcodes_pkey" PRIMARY KEY, btree (gid)
"zipcodes_geom_idx" gist (geom)
```

Step three: An example query

```
--
-- getting a zipcode based upon the LAT/LON
--
gis_demo=# select * from gis.findzipcode(-87.798615,30.53711);
  longitude | latitude | zipcode
------------+----------+---------
 -87.798615 | 30.53711 | 36576
```

On the road to mastering GIS

There are a couple of things to keep in mind when querying PostGIS-powered databases:

- Follow the PostGIS reference documentation closely: it's easy to read, explains many of the ideas behind the function call, and provides many examples.
- The GIS standard requires that functions be documented using mixed case.
- GIS function calls, implemented in Postgres, are written using lowercase.
- For the adventurous, here's a comprehensive and freely available dataset of shapefiles for the city of Kamloops, British Columbia.

Because so much of our modern big data insights depend upon raw GIS data, directly querying a GIS database empowers one to create even more powerful and precise insights. PostGIS is an excellent way for the DEV, DBA, and SRA to learn all things GIS. Have Fun!

Bonus chapter: Cost optimization

This bonus chapter is dedicated to reducing PostgreSQL costs in the cloud and is pulled from both the Percona blog and a comprehensive webinar our experts did on the subject.

Managing costs while maintaining optimal database performance is a challenge many organizations face, and this chapter provides a roadmap for minimizing costs without sacrificing the database performance your business relies on.

It begins with the principle of usage efficiency—allocating resources precisely according to need without over-provisioning — and continues with insights into effective monitoring with Percona Monitoring and Management, understanding AWS's guidance on over-provisioned instances, and exploring various cost-saving measures like instance re-sizing.

If you've been tasked with reducing your PostgreSQL costs, this is the chapter you have been waiting for!

Reducing PostgreSQL Costs in the Cloud

By Mario García

Mario worked as a Technical Evangelist at Percona until 2023. He has been an active open source contributor for over 15 years.

If you're using PostgreSQL in the cloud, there's a good chance you're spending more than you need in order to get the results you need for your business.

Let's take a look at how to get the benefits you need while spending less, based on the recommendations presented by Dani Guzmán Burgos, our Percona Monitoring and Management (PMM) Tech Lead, on this webinar (now available on demand) hosted in November last year.

Usage reduction: What to look for to reduce PostgreSQL cloud costs

The first step in cost reduction is to use what you need and not more. Don't pay for capacity you don't use or need.

Usage reduction is a continuous process. Identifying which resources you can trim to reduce your monthly bill can be difficult, but looking at the right metrics will help you understand the application's actual requirements.

In the Home Dashboard of PMM, a low CPU utilization on any of the database services that are being monitored could mean that the server is inactive or over-provisioned. Marked in red in Figure 1 is a server with less than 30% of CPU usage. PMM can also show you historical data that can help you identify how long a service has been in a given state. Configuration of the CPU metrics can be changed in the dashboard. These color-coded states on panels are available in PMM 2.32.0 and later.

From the Amazon Web Services (AWS) documentation, an instance is considered over-provisioned when at least one specification of your instance, such as CPU, memory, or network, can be sized down while still meeting the performance requirements of your workload, and no specification is under-provisioned.

Over-provisioned instances may lead to unnecessary infrastructure costs.
Making the use of resources efficiently and ensuring that this does not impact the budget available for cloud computing is not a one-time fix but a continuous cycle of picking properly sized resources and eliminating over-provisioning.

Use reduction at scale requires a cultural shift, and engineers must consider the cost as they think of memory or bandwidth as another deployment KPI.

Think of a gaming company that creates a game that is getting popular, so the number of resources needed to support more users would increase considerably. But if the game loses popularity, the server would become over-provisioned, and the resources allocated must be re-sized to better fit the application's needs.

Re-sizing to save costs

There are three approaches to usage reduction:

- Reducing waste
- Re-architecting
- Re-sizing to what you need

Regardless of the method you're using to deploy your PostgreSQL instance, here are some metrics that would determine when re-sizing is needed:

- CPU utilization
- Memory usage
- Network throughput
- Storage usage

Remember that optimizing your infrastructure is intended for more than cost savings. You have to ensure that the operation is not impacted when you're making decisions based on the metrics. The primary goal is to ensure that the services themselves do not run out of the required operating capacity.

PostgreSQL in the cloud

CPU

Considering AWS as your cloud platform of choice, the configuration made for your infrastructure will influence the performance of your application and monthly costs. For example, an Amazon Elastic Compute Cloud (EC2) instance with Graviton2 processors will be a better choice than non-ARM options, as it's cheaper, and you will get real and faster cores, which means the CPU cores are physical and not with hyper-threading. Graviton2 processors aim to use Reserved Instances to save costs in the long run.

Benefits of Graviton2 Processors

- Best price performance for a broad range of workloads
- Extensive software support
- Enhanced security for cloud applications
- Available with managed AWS services
- Best performance per watt of energy used in Amazon EC2

Storage

Continuing with the AWS example, choosing the right storage option will be key to performance. Amazon Elastic Block Store (EBS) is your good-to-go option for disk space.

From AWS documentation, Amazon EBS is an easy-to-use, scalable, high-performance block-storage service designed for Amazon EC2.

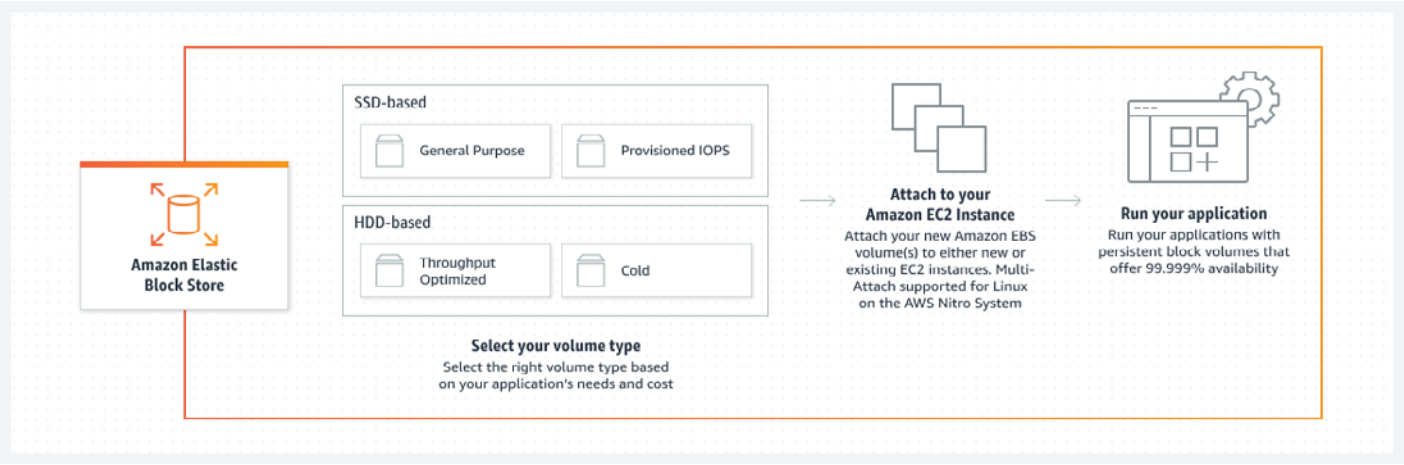


Figure 2: Amazon Elastic Block Storage

Running relational or NoSQL databases is one of the use cases where EBS is recommended. You can deploy and scale your databases, including SAP HANA, Oracle, Microsoft SQL Server, PostgreSQL, MySQL, Cassandra, and MongoDB.
With EBS, you can configure HDD-based volumes optimized for large streaming workloads or SSD-based volumes (recommended for database workloads) optimized for transactional workloads.

An SSD volume can be any of the following types:

- io1
- io2
- io2 Block Express
- gp2
- gp3

Which one is a better choice for storage? It will depend on the requirements of your workload, including disk space, input/output operations per second (IOPS), and throughput rate (MB/s). Your configuration must be cost-optimized as well.

Avi Drabkin's blog post is a recommended reading on this matter, as he analyzes the configuration required for every volume type to satisfy the requirements of a particular use case. For more information on EBS volume types, check the Amazon EBS Volume Types page.
Multi-AZ deployments vs. read replicas

Multi-AZ deployment

In an Amazon RDS Multi-AZ deployment, Amazon RDS automatically creates a primary database (DB) instance and synchronously replicates the data to an instance in a different AZ. Amazon RDS automatically fails over to a standby instance without manual intervention when it detects a failure.

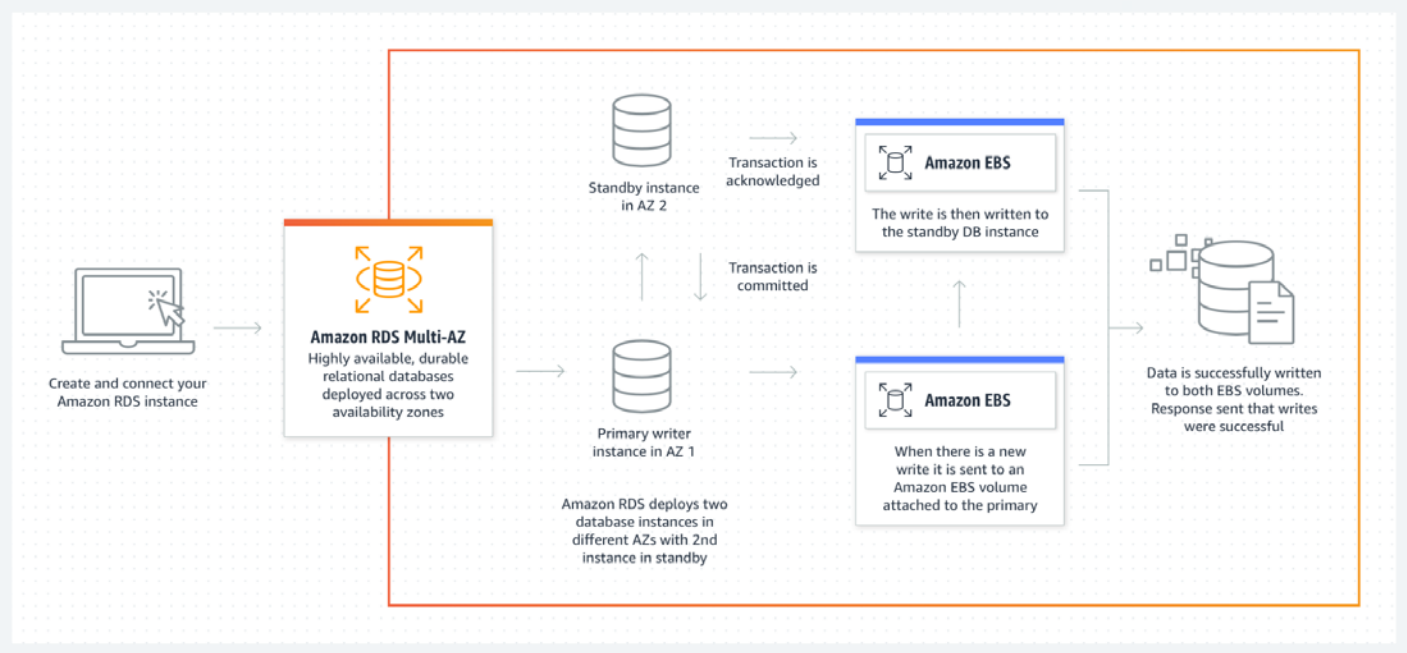
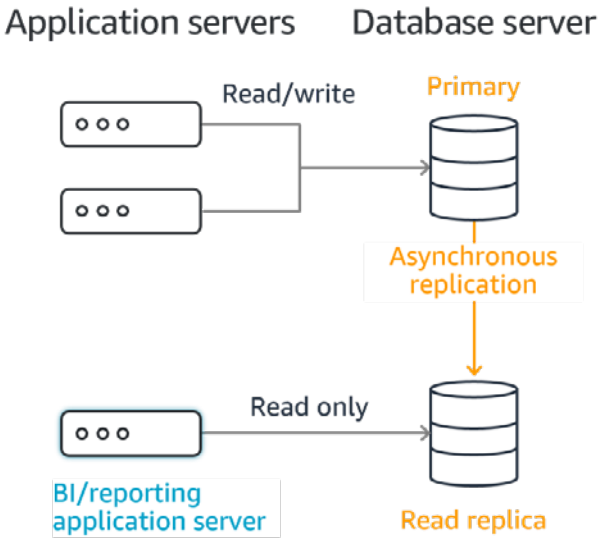


Figure 3: Amazon RDS Multi-AZ Deployment

Read replica

Amazon RDS creates a second DB instance using a snapshot of the source DB instance. It then uses the engines' native asynchronous replication to update the read replica whenever there is a change to the source DB instance. The read replica operates as a DB instance that allows only read-only connections; applications can connect to a read replica just as they would to any DB instance. Amazon RDS replicates all databases in the source DB instance.



Which option is better?

Multi-AZ deployments offer advantages, especially for HA and disaster recovery. The trade-off is that multi-AZ deployments are expensive.

A better option would be to deploy reader instances and combine them with the use of a reverse proxy, like pgpool-II or pgbouncer. The reader instances also cost more than a standard setup, but you can use them for production to handle everyday database traffic.

Pgpool-II can be used not only for reducing connection usage, which will be helpful in reducing CPU and memory usage but also for load balancing. With load balancing, you can redistribute the traffic, sending the reading requests to your read replicas and writing requests to your main database instance automatically.

Regarding read replicas, in AWS, you cannot promote an RDS PostgreSQL read replica, which means a read replica can't become the primary instance. Whenever you try to do this, the read replica detaches from the primary instance and become its own primary instance, and you will end up having two different clusters.

One solution is using the pglogical extension for creating replicas outside the RDS path. When combining the pglogical replication with a reverse proxy, you will still get the benefits of a managed database, including backups, minor upgrades maintenance, recovery support, and being tied to the Multi-AZ configuration, which translates to full control over planned failovers.

Also, converting a replica to the primary instance would be a better upgrade approach. For example, if you need to upgrade a database with a large amount of data, this process could take hours, and your instance won't be available during this time. So, with this configuration, you can upgrade a replica and later convert that replica to the primary instance without interrupting operations.

Vacuum

Bloating in the database is created when tables or indexes are updated; an update is essentially a delete-and-insert operation. The disk space used by the delete is available for reuse but not reclaimed, creating the bloat.

How to remove bloat? That's what the vacuum process is intended for with the help of autovacuum and vacuumdb.

Autovacuum is a daemon that automates the execution of VACUUM and ANALYZE (to gather statistics) commands. Autovacuum checks for bloated tables in the database and reclaims the space for reuse.

vacuumdb is a utility for cleaning a PostgreSQL database. vacuumdb will also generate internal statistics used by the PostgreSQL query optimizer. vacuumdb is a wrapper around the SQL command VACUUM.

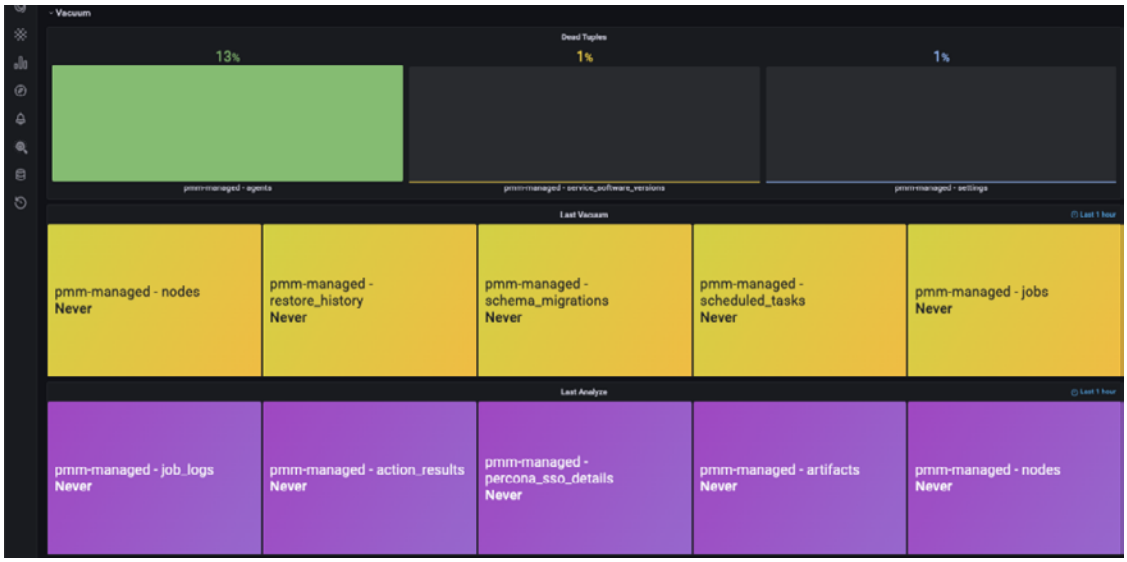
Running autovacuum on a database with tables that store tens of terabytes can become such a big overhead, not only for the amount of data but the dead tuples generated on every transaction. A solution could be changing the EBS volume type for an io1 that can support this workload, but your monthly bill would also increase.

A better solution for this scenario would be running the vacuum process on demand. Disabling autovacuum doesn't mean not running it at all. You just run it on demand on lower-traffic hours where the overhead on the disk performance doesn't affect the operation. The overhead of running with a high percentage of dead tuples is cheaper in that case.

This way, you guarantee that the resources will be available during the day when most operations occur.

Is this the best solution for every scenario? No, it will depend on the requirements of your database.

Monitoring your database for dead tuples (bloat) is also recommended. For this matter, you can use the Experimental PostgreSQL Vacuum Monitoring. This experimental dashboard is not part of PMM, but you can try it and provide feedback.



What about serverless?

With serverless, you truly pay only for what you're actively using, and unused resources aren't typically easy to be left flying around, but the move to serverless isn't without cost. The complexity of building any migration plan to serverless resides in execution and has very little to do with cost savings. There's an entirely different lens through which you can evaluate serverless: total cost of ownership (TCO).

The TCO refers to the cost of engineering teams that are required to build a solution and the impact of time to market on the success and profitability of a service. Serverless allows you to delegate a lot of responsibility to the cloud provider.

Duties that DevOps engineers would typically perform (server management, scaling, provisioning, patching, etc.) become the responsibility of AWS, GCP, or Azure. It leaves dev teams with free time to focus on shipping

differentiated features faster.

With TCO, you must consider that people's costs may cancel out any infrastructure savings when considering moving from a monolithic application to a serverless one.
Returning to the benefits versus effort, you should consider the overall cost of redesigning services for serverless against the potential for reducing costs.

Conclusion

Knowing your project's database requirements will be essential when choosing the services and hardware configuration you will pay your cloud service provider for. The configuration you make must guarantee the proper functioning of your application and will determine the monthly costs.

The number of resources required may vary over time, and metrics like CPU usage, memory usage, and disk storage will help you determine when re-sizing your infrastructure is needed.

For example, if the number of database transactions decreases after a certain period of time, you will have more resources than you need, and it will be important to change the configuration of your infrastructure to guarantee the new requirements and pay what is really being used.

Conclusion

As we wrap up PostgreSQL Performance Tuning, we hope you have gained some of the knowledge and tools necessary to navigate the complexities of PostgreSQL optimization. Each chapter, curated with expertise from Percona's seasoned professionals, has provided insight into such advanced topics as partitioning, schema design, query optimization... and everything in between. While it's not a comprehensive guide to all things PostgreSQL performance, the strategies inside will help you manage and maintain a PostgreSQL environment that is optimized for peak performance and aligned with the principles of cost efficiency and scalability.

If you're looking to deepen your understanding of database optimization and seeking expert advice on PostgreSQL, we invite you to subscribe to our blog. Here, you'll find an extensive collection of insights, tips, and comprehensive guides written to assist you in mastering database management and optimization specific to PostgreSQL.

Additionally, if you're interested in a thorough evaluation of your PostgreSQL database's health and performance, we recommend visiting our Database Performance Assessment page. Choosing a Database Wellness Exam with us not only helps you sidestep the significant expenses linked to suboptimal database performance but also connects you with a dedicated team committed to ensuring your database environment is finely optimized.

Databases Run Better
with Percona

percona.com