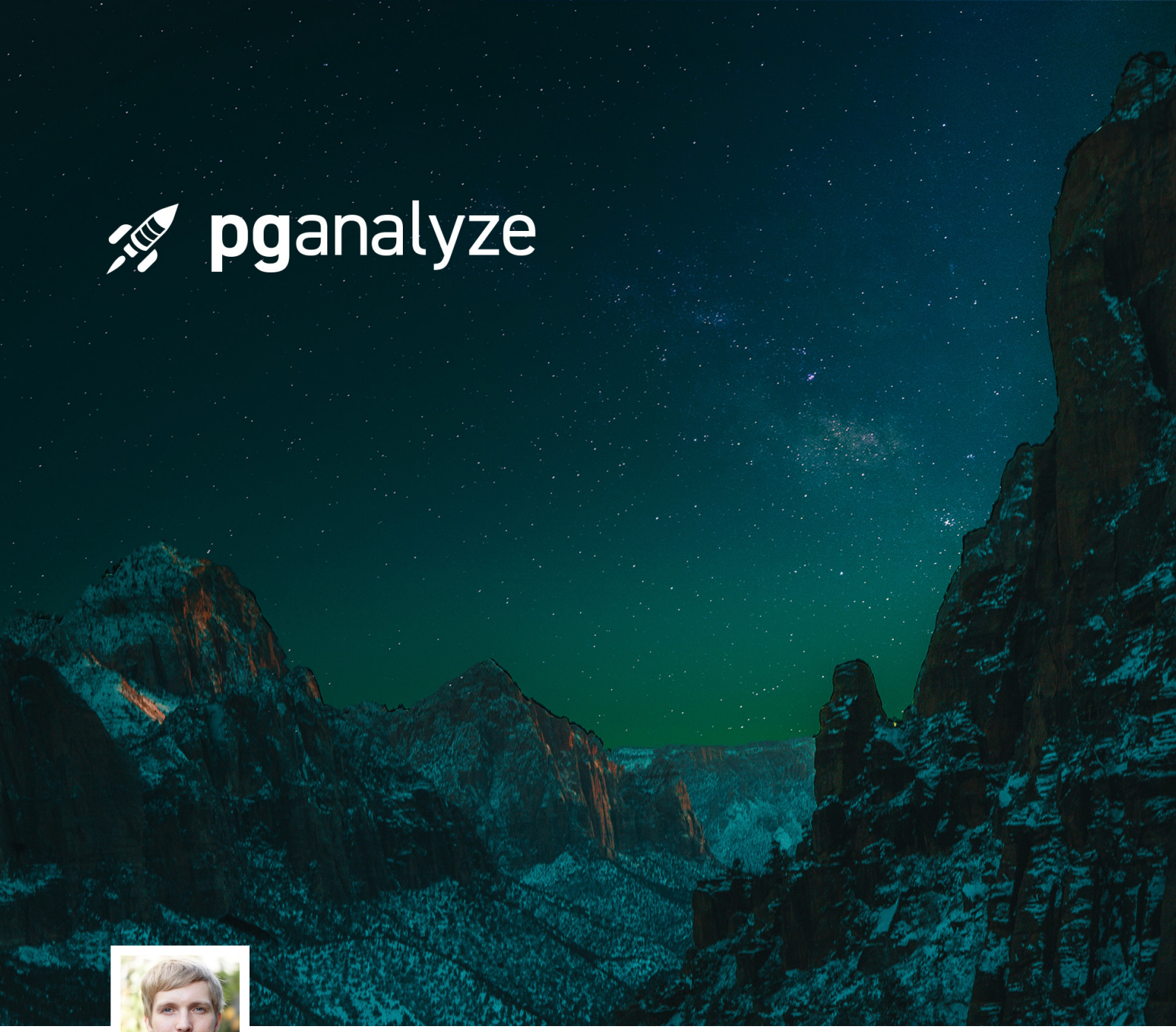


pganalyze

LUKAS FITTL
FOUNDER AT PGANALYZE

The Most Important Events to Monitor in Your Postgres Logs

About the Author.

Lukas Fittl is the founder of pganalyze. His fascination with technology has always been combining deep technical know-how with usable interfaces and great design.

Lukas has had his fair share of scaling experience, most notably co-founding the blogging network Soup.io, and taking responsibility for scaling the PostgreSQL-based backend to more than 50,000,000 posts.

He is a frequent speaker on topics around Agile and Lean Product Management, and has a personal mission to distribute product ownership & the customer's perspective into engineering teams.

DBAs and developers use pganalyze to identify the root cause of performance issues, optimize queries and to get alerts about critical issues.

Learn more about pganalyze here.

The Most Important Events to Monitor in Your Postgres Logs

Reading and understanding your logs is one of the most essential things you can do to get visibility into what's going on in your database.

In this book, we'll highlight a few perspectives on how Postgres logs can be useful, how you can extract `EXPLAIN` plans from your logs, and how to know about critical issues ahead of time.

Such insights are not only interesting, they help prevent major issues (see what happened to Sentry further down below) and will turn into valuable business benefits along the way.

In this eBook, we will walk you through the Top 6 Postgres log events for monitoring query performance and preventing database downtime.

ABOUT

Our book will cover the most important events to:

- Prevent Downtime
- Reduce I/O Spikes
- Improve Query Performance
- Deep Diving Into Performance Issues
- Handling secret data and PII contained in Postgres logs

The events we will look in detail at are:

- Transaction ID (TXID) Wraparound
- Data Corruption
- Checkpoints
- Temporary Files
- Lock Notices & Deadlocks
- EXPLAIN plans through auto_explain

Without further ado, let's get started!

Preventing Downtime: Critical Events To Know About

The thing everyone cares about and fears the most: What could take my database down, and make it unavailable? Postgres logs actually contain significant amount of information in the case of potential issues.

Two critical events that most seasoned database administrators are familiar with: **Transaction ID (TXID) Wraparound** and **Data Corruption**.

Transaction ID (TXID) Wraparound

The way Postgres supports multiversion concurrency control (MVCC), is by tracking each transaction internally using a 32-bit transaction ID, resulting in a maximum of a bit over 4 billion unique transaction IDs.

You may ask why they haven't increased it to 64-bit yet. It's complicated, there are efforts under way, but for now we have to work with the status quo. And that means if you have a highly active database, you can run out of available transaction IDs, resulting in the database becoming unavailable.

To protect from the obvious case (if we count up, at some point we reach the maximum), there is the functionality of a frozen transaction ID, reflecting a row in the database that is visible to all currently active and future transactions, for which we don't need to track a unique transaction ID anymore. This is implemented by the `VACUUM` process, commonly run by the autovacuum daemon. If you have well-tuned autovacuum, you should not run into transaction ID wraparound. However, if you do not, you may see the following event in your Postgres logs:

```
1   WARNING: database "template1" must be vacuumed within 938860 transactions
2   HINT: To avoid a database shutdown, execute a database-wide VACUUM in that database.
3   You might also need to commit or roll back old prepared transactions.
```

As the message says, if no action is taken, the database will shut down and become unavailable, to avoid inconsistencies - we can't assign an already used transaction ID to a new transaction that has started, as that would cause inconsistent visibility in the MVCC model.

This type of issue is common enough that popular services such as Sentry have written posts about their incidents:

> *"On Monday, July 20th, Sentry was down for most of the US working day. [...] Once Postgres kicks in its XID wraparound protection, unless you're willing to accept data loss, your only option is to stop accepting writes and vacuum the relations."*

David Cramer, Co-Founder and CEO, Sentry - Transaction ID Wraparound in Postgres

In pganalyze, we classify this log event as "A61: Database must be vacuumed within N transactions (TXID Wraparound Warning)", as well as "A62: Database is not accepting commands to avoid wraparound data loss".

In particular, for the latter log event your only choice is to shut down the database completely, enter single-user mode, and run a manual vacuum. Monitoring both of these log events, as well as tuning your autovacuum, are therefore critical parts of operating a production Postgres database.

Data Corruption

Another well known log event that can lead to a service becoming unavailable, typically seen in larger database setups, is related to data corruption. First of all, make sure you have checksums enabled in Postgres - otherwise you will

never know when data becomes corrupted.

Data Corruption log events can have many different sources of origin, for example a bad backup used as a source for a new database, or actual failures of the underlying disk. In any case, when you see this message in your Postgres logs, you know something is wrong:

```
1   WARNING: page verification failed, calculated checksum 20919 but expected 15254
2   ERROR: invalid page in block 335458 of relation base/16385/99454
3   STATEMENT: SELECT * FROM my_table
```

At this point you have three essential choices:

1. **Fail over to an HA standby server / follower database**
   If you have a follower or standby database, that has a copy of the full data, and the underlying corruption is due to a hardware problem (i.e. disk failure), there is a good chance that the standby will not have the problem

2. **Replay WAL from an old enough base backup**
   Assuming you store both your base backups and all WAL (Write-Ahead Logging) files between them,

you should be able to make a copy/fork of your database from an old base backup and then replay the WAL all the way to the current timestamp. With this approach, as well as (1), it's important to remember that base backups will copy any corrupted pages, but the WAL stream has a clean version of the data as it was written.

3. **Accept that some data is lost, and let Postgres move on**
   In some scenarios this may be the only choice you have, and your primary goal is to let Postgres continue its work without constantly returning errors.

You can read more about this log event on our documentation page for S6: Invalid page / page verification failed (data corrupted), as well as a more detailed guide on how you can have Postgres accept corrupted data in order to be able to read it out.

Reducing I/O spikes: Tuning checkpoints

Moving on from critical issues, to something less critical, but nonetheless important: Avoiding spikes in I/O operations that Postgres performs in the background.

You may be familiar that Postgres has multiple processes that run in the background, in addition to the processes that run for each connection. One of these processes is the checkpointer process, responsible for writing data thats is only in shared memory and the WAL, to disk.

In the case of a crash, the checkpoint written by this process acts as the last known safe point, from which the WAL gets replayed up until the point of the crash.

Now, if you only optimized for fast recovery after crashes, you would want a lot of checkpoints, to keep the amount of WAL that needs to be replayed low. However, in reality, the bigger issue related to checkpoints is the fact that they are quite I/O intensive, as they need to write a lot of data to disk and run fsync to ensure crash safety.

The first step is to understand when a checkpoint runs, which you can see by enabling the `log_checkpoints` configuration option in Postgres. You will then see log messages that look like this:

```
LOG: checkpoint starting: xlog
```

As well as this, a bit later:

```
LOG: checkpoint complete: wrote 111906 buffers (10.9%); 0 WAL file(s) added, 22 removed,
29 recycled; write=215.895 s, sync=0.014 s, total=216.130 s; sync files=94, longest=0.014 s,
average=0.000 s; distance=850730 kB, estimate=910977 kB
```

The start message tells you a key point: **Why was this checkpoint performed?**

There are two major reasons for checkpoints in Postgres during normal operations:

- "xlog" - Checkpoint that runs after a certain amount of WAL has been generated since the last checkpoint, as determined by `max_wal_size` (or `checkpoint_segments` in older Postgres versions)

- "time" - Checkpoint that runs after a certain amount of time has passed since the last checkpoint, as determined by `checkpoint_timeout`

The common tuning guidance is to optimize for more **time** based checkpoints. This type of checkpoints is written gradually over time (as controlled by `checkpoint_completion_target`), therefore reducing the I/O impact at a given moment, and making a predictable pattern that complements the actual workload.

On the other hand, a **xlog** based checkpoint is forced to work as quickly as possible, therefore leading to a large I/O spike at the time it is started, until it is finished.

In order to have more "time" based checkpoints you can increase the `max_wal_size` setting, therefore allowing more WAL to be retained until a checkpoint is required. This comes at the expense of longer recovery times after crashes, so it is recommended to do some restore tests when adjusting this setting.

In pganalyze, we detect checkpoint behaviour as W40: Checkpoint starting as well as W41: Checkpoint complete. You may also see W42: Checkpoints occurring too frequently:

```
1   LOG: checkpoints are occurring too frequently (18 seconds apart)
2   HINT: Consider increasing the configuration parameter "max_wal_size".
```

This is essentially a built-in hint into Postgres when there are too many "xlog" based checkpoints. In any scenario other than bulk data loads, you should look at tuning the settings if you see this event.

Improving Query Performance: Temporary Files, Locks & More

Moving on from issues you see on the operational side to the kind of log events that impact application and query performance more directly, and may sometimes be seen as either CPU or I/O spikes.

Temporary Files

Postgres has many uses for temporary files, but the most important one for performance is the situation where Postgres works with a large result set that needs to be sorted or grouped. In case of the process not being assigned enough memory, a temporary file gets created, written to, and read from, in order to perform the sorting or grouping operation.

By default you won't notice this, unless you run `EXPLAIN ANALYZE` on a query to see the detailed query execution plan. However, you can enable the `log_temp_files` parameter, which specifies the threshold

after which the creation of a temporary file gets logged. For most systems, with the exception of a heavily used data warehousing setup, its reasonable to set `log_temp_files = 0`, which logs the creation of all temporary files.

You will then see a log notice like this during query execution:

```
1   LOG: temporary file: path “base/pgsql_tmp/pgsql_tmp15967.0”, size 200204288
2   STATEMENT: alter table pgbench_accounts add primary key (aid)
```

In the case of a DDL statement, like above, you may choose to not optimize this further, but simply take it as an explanation of an I/O spike visible in graphs. For DDL operations you can tune the `maintenance_work_mem` setting to keep operations in memory.

However, in the case of `SELECT` statements, this can be more critical for performance. This problem is increased by the fact that Postgres ships with a very low default for the `work_mem` setting that is used for regular operations. By default this only allows for **4 MB of memory** used for sorting and grouping operations.

In pganalyze, this event is visible as S7: Temporary file: path ..., size ... (temp file created). If you see

a lot of temporary file log events, we recommend increasing `work_mem`. You can also set this on a per-connection, or per-statement basis, using the `SET` command.

You may want to combine a `SET` command first, together with `EXPLAIN ANALYZE`, to verify the exact temporary file creation behaviour.

Lock Notices

Another common source of performance issues is the locking that happens behind the scenes in Postgres. Every time you modify a row, a certain lock needs to be taken to avoid concurrent updates at the exact time. Similarly, for more complex operations such as changing the table schema, more aggressive locks will be taken to avoid any concurrent modifications with an inconsistent state.

To understand better when a query has to wait for a lock request to be granted, you can enable the `log_lock_waits = on` setting. By default **this will log all lock requests that were not granted within 1 second**, the default deadlock detector timeout, which is utilized to provide this analysis.

Whilst a query is waiting for a lock, you will see L71: Process still waiting for lock on tuple / relation / object:

```
LOG: process 2078 still waiting for ShareLock on transaction 1045207414 after 1000.100 ms
DETAIL: Process holding the lock: 583. Wait queue: 2078, 456
QUERY: INSERT INTO x (y) VALUES (1)
CONTEXT: PL/pgSQL function insert_helper(text) line 5 at EXECUTE statement
STATEMENT: SELECT insert_helper($1)
```

Once the lock has been granted, you will see L70: Process acquired lock on tuple / relation / object:

```
LOG: process 583 acquired AccessExclusiveLock on relation 185044 of database 16384 after
2175.443 ms
STATEMENT: ALTER TABLE x ADD COLUMN y text;
```

There is yet another category of lock notice, the deadlock notice, aka L73: Deadlock detected (transaction rolled back):

```
LOG: process 123 detected deadlock while waiting for AccessExclusiveLock on extension
of relation 666 of database 123 after 456.000 ms
ERROR: deadlock detected
DETAIL: Process 9788 waits for ShareLock on transaction 1035; blocked by process 91.
Process 91 waits for ShareLock on transaction 1045; blocked by process 98.
Process 98: INSERT INTO x (id, name, email) VALUES (1, 'ABC', 'abc@example.com') ON
CONFLICT(email) DO UPDATE SET name = excluded.name, /* truncated */
Process 91: INSERT INTO x (id, name, email) VALUES (1, 'ABC', 'abc@example.com') ON
CONFLICT(email) DO UPDATE SET name = excluded.name, /* truncated */"
HINT: See server log for query details.
CONTEXT: while inserting index tuple (1,42) in relation "x"
STATEMENT: INSERT INTO x (id, name, email) VALUES (1, 'ABC', 'abc@example.com') ON
CONFLICT(email) DO UPDATE SET name = excluded.name RETURNING id
```

In a situation like that, Postgres is able to automatically recover by canceling one of the two offending transactions, to avoid a deadlock.

In general, lock issues can be difficult to analyze, but especially when looking at a pattern over time, or after an incident, it is critical to run your system with `log_lock_waits = on` for full details in the Postgres logs.

Deep Diving Into Performance Issues: Get query plans using auto_explain

Now, for the holy grail of performance tuning using the Postgres logs: Automatic `EXPLAIN` of query plans. If you are not familiar with `EXPLAIN`, it provides a mechanism to show you how exactly Postgres will execute a query, giving details on how much data was loaded, which kind of index was used, and whether a sort required a temporary file on disk.

Following, let's have a look at the `EXPLAIN` output in detail to understand how Postgres executes our query:

```
=> EXPLAIN (ANALYZE, BUFFERS) SELECT ...
QUERY PLAN
-----------------------------------------------------------------------------------------------
 Limit  (cost=34005.21..34005.22 rows=1 width=1887) (actual time=4898.458..4899.708
rows=5 loops=1)
   Buffers: shared hit=83992 read=43725 dirtied=642
   I/O Timings: read=12409.188
   ->  Sort  (cost=34005.21..34005.22 rows=1 width=1887) (actual time=4898.457..4898.458 rows=5 loops=1)
         Sort Key: query_samples.occurred_at DESC
         Sort Method: top-N heapsort  Memory: 45kB
         Buffers: shared hit=83992 read=43725 dirtied=642
         I/O Timings: read=12409.188
         ->  Gather  (cost=14098.53..34005.20 rows=1 width=1887) (actual time=2497.891..4898.888 rows=481
 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               Buffers: shared hit=83989 read=43725 dirtied=642
               I/O Timings: read=12409.188
               ->  Nested Loop  (cost=13098.53..33005.10 rows=1 width=1887) (actual
time=2495.670..4894.721 rows=160 loops=3)
                     Buffers: shared hit=83989 read=43725 dirtied=642
                     I/O Timings: read=12409.188
                     ->  Parallel Bitmap Heap Scan on query_explains
(cost=13097.97..29916.16 rows=360 width=1879) (actual time=833.370..4005.660 rows=8204 loops=3)
                           Recheck Cond: ((query_fingerprint = ANY ('{...}'::bytea[]))
AND (database_id = '6'::bigint))
                           Filter: (postgres_role_id = '...'::uuid)
                           Heap Blocks: exact=7975
                           Buffers: shared hit=87 read=25495 dirtied=347
                           I/O Timings: read=9962.788
                           ->  BitmapAnd  (cost=13097.97..13097.97 rows=4592 width=0)
(actual time=829.769..829.769 rows=0 loops=1)
                                 Buffers: shared hit=15 read=1721
                                 I/O Timings: read=724.721
                                 ->  Bitmap Index Scan on index_query_explains_on_
query_fingerprint  (cost=0.00..1042.22 rows=24183 width=0) (actual time=76.491..76.491 rows=25610 loops=1)
                                       Index Cond: (query_fingerprint = ANY ('{...}'::bytea[]))
                                       Buffers: shared hit=9 read=153
                                       I/O Timings: read=70.627
                                 ->  Bitmap Index Scan on index_query_explains_on_
database_id  (cost=0.00..12055.07 rows=433418 width=0) (actual time=747.779..747.779
rows=454882 loops=1)
                                       Index Cond: (database_id = '6'::bigint)
                                       Buffers: shared hit=6 read=1568
                                       I/O Timings: read=654.094
                     ->  Index Scan using query_samples_pkey on query_samples  (cost=0.56..8.58
rows=1 width=24) (actual time=0.107..0.107 rows=0 loops=24612)
                           Index Cond: (query_sample_id = query_explains.query_sample_id)
                           Filter: ((occurred_at >= '2019-04-21 08:54:20.136816'::timestamp without
time zone) AND (occurred_at <= '2019-04-22 08:54:20.136889'::timestamp without time zone))
                           Rows Removed by Filter: 0
                           Buffers: shared hit=83902 read=18230 dirtied=295
                           I/O Timings: read=2446.400
 Planning time: 0.386 ms
 Execution time: 4900.373 ms
```

Typically, you would run the `EXPLAIN` command on a query you are actively debugging and whose performance you are trying to improve. However, often a query plan can change over time, as Postgres updates its statistics and the table contents change.

This is where the auto_explain extension comes in. Its bundled with Postgres, similar to `pg_stat_statements` (which is used for aggregate query performance information in tools such as pganalyze), and needs to be explicitly added to the `shared_preload_libraries` setting. A common configuration would therefore be something like this:

```
1   shared_preload_libraries = pg_stat_statements, auto_explain
```

In addition, `auto_explain` provides multiple tunables to configure the exact behaviour. Our recommended configuration is this:

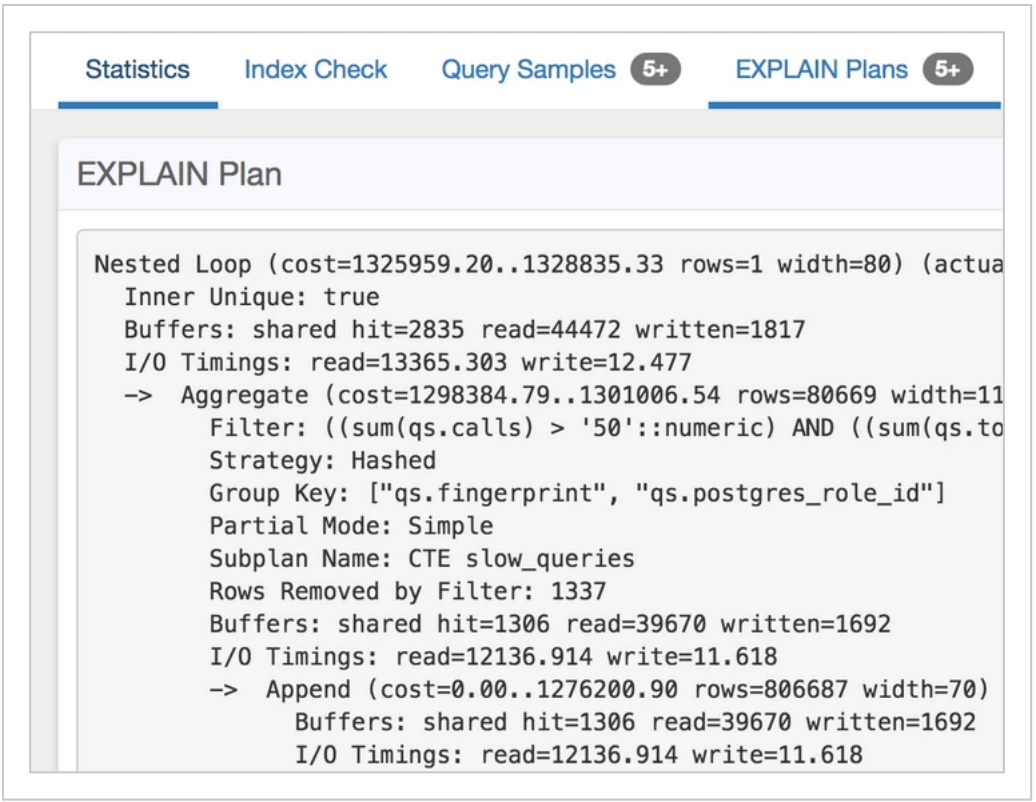
```
auto_explain.log_analyze = 1
auto_explain.log_buffers = 1
auto_explain.log_timing = 0
auto_explain.log_triggers = 1
auto_explain.log_verbose = 1
auto_explain.log_format = json
auto_explain.log_min_duration = 1000
auto_explain.log_nested_statements = 1
auto_explain.sample_rate = 1
```

Note the disabling of `log_timing`, which is expensive, and should only be enabled on non-production systems. If you see any negative impact on query performance after enabling `auto_explain`, you may also want to disable `log_analyze`, in particular when running on older hardware. You can find a more detailed setup guide in the pganalyze documentation.

Now, once enabled, when a query runs that exceeds the threshold (1 second in the above configuration), you will see an event like this in the Postgres log:

```
1   LOG: duration: 1681.452 ms  plan:
2     Query Text: UPDATE pgbench_branches SET bbalance = bbalance + 2656 WHERE bid = 59;
3     Update on public.pgbench_branches  (cost=0.27..8.29 rows=1 width=370) (actual rows=0
4   loops=1)
5       Buffers: shared hit=7
6       ->  Index Scan using pgbench_branches_pkey on public.pgbench_branches
7   (cost=0.27..8.29 rows=1 width=370) (actual rows=1 loops=1)
8             Output: bid, (bbalance + 2656), filler, ctid
9             Index Cond: (pgbench_branches.bid = 59)
```

Whilst helpful to look at in the logs, it can be a bit cumbersome to work with. We've therefore developed an integration for pganalyze, where you can see the `EXPLAIN` plans from `auto_explain` directly on each query page:



Handling secret data and PII contained in Postgres logs

One critical detail that you may find when working with the Postgres logs: They contain a lot of detailed information, sometimes even excerpts of data from the actual tables in the database, that may contain PII or PHI.

Such data is often restricted in terms of who should be able to see it within an organization, as well as which third-party services (if any) should receive a copy of the data.

Here is one example of a common log event that may contain sensitive data, V100: Duplicate key value violates unique constraint:

```
1   ERROR: duplicate key value violates unique constraint “test_constraint”
2   DETAIL: Key (b, c)=(12345, mysecretdata) already exists.
3   STATEMENT: INSERT INTO a (b, c) VALUES ($1,$2) RETURNING id
```

As you can see the actual table data, “12345” and “mysecretdata” in this case, was output in the Postgres logs. This should obviously not be shared

broadly outside of the actual database server.

Another common example is the syntax error log event. We've all mis-typed a SQL command once, but unfortunately many of us have also pasted text into the wrong terminal window, such as internal admin passwords or connection URLs. Assuming the pasted text is not valid SQL, you will then see something like this in the logs:

```
1   ERROR:  syntax error at or near "postgres" at character 1
2   STATEMENT:  postgres://username:password@host/db
```

As you can see, we might have accidentally sent (and stored) an internal password in the Postgres logs.

When we look at the information visible in the Postgres logs, we can distinguish the following types:

1. **Log event text:** The actual text for a log event such as "checkpoint starting". This can be safely sent to a service and stored for later viewing.

2. **Credentials:** Passwords and other credentials, e.g. private keys. This should typically not be stored.

3. **Parsing error:** User supplied text during parsing errors – could contain anything, including credentials. This should typically not be stored.

4. **Statement text:** All statement texts, which may contain table data if not using bind parameters. Depending on your system configuration this may be safe to store.

5. **Bind parameters:** Bind parameters for a statement, which may contain table data for `INSERT` statements. This is should usually not be stored.

6. **Table data:** Table data contained in constraint violations and `COPY` errors. This should not be stored.

7. **Operational errors:** System, network errors, file locations and configured commands, e.g. archive command. This is typically safe to store, but care needs to be taken when credentials are directly specified in configuration settings such as `archive_command`.

8. **Unidentified text:** Log output that could not be identified and might contain secrets

As you can see, it can be difficult to work with Postgres logs safely, and sharing them with your broader team for reliable operations and performance improvements.

To help you with this goal of better collaboration

within the team, we've implemented a log filtering system in pganalyze that you can utilize to mask some of this data. For example, we would typically recommend the following setting for the pganalyze collector:

```
filter_log_secret: credential, parsing_error, table_data
```

This will replace all known credentials, parsing errors and table data with a replacement character ("X"). With that configuration our initial examples turn into:

```
ERROR: duplicate key value violates unique constraint "test_constraint"
DETAIL: Key (b, c)=(XXXXXXXXXXXXXXXXXXX) already exists.
STATEMENT: INSERT INTO a (b, c) VALUES ($1,$2) RETURNING id
```

This is log output we can safely share with our broader team, allowing everyone, including application developers, to safely work with the Postgres logs and understand the application behaviour and root cause of performance issues better.

Conclusion

In this book, we've looked at the most important log events to monitor in your Postgres database.

In particular, we dug deeper on 6 events the team here at pganalyze considers especially important as these events help among others, prevent downtime, reduce I/O spikes, improve query performance, and more.

Are there any events you think we missed or that you and your team deem especially important? We'd love to hear your thoughts.

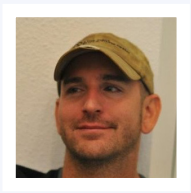
In any case, you can always feel free to reach out to us with any questions or feedback on our pganalyze contact page.

Try pganalyze for free

pganalyze can save you and your development team many hours spent debugging database performance, and lets you spend that time on strategic efforts and application development instead.

Get started easily with a free 14-day trial, or learn more about our Enterprise product.

If you want, you can also request a personal demo.



"The scale and volume of data we handle meant that, prior to using pganalyze, I had no easy visibility into this information. pganalyze has saved me at least a day of forensic analysis when debugging database problems."

Jon Erdman, Senior Postgres DBA
Bitbucket Cloud, Atlassian

About pganalyze.

DBAs and developers use pganalyze to identify the root cause of performance issues, optimize queries and to get alerts about critical issues.

Our rich feature set lets you optimize your database performance, discover root causes for critical issues, get alerted about problems before they become big, gives you answers and lets you plan ahead.

Hundreds of companies monitor their production PostgreSQL databases with pganalyze.

Be one of them.
Sign up for a free trial today!