LUKAS FITTL
FOUNDER AT PGANALYZE

Best Practices for Optimizing Postgres Query Performance

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.

Best Practices for Optimizing Postgres Query Performance

Over the last 5 years, we've learned a lot on how to optimize Postgres performance. In this eBook, we wrote down our key learnings on how to get the most out of your database.

Have you ever received questions from your team asking why your product's application is running slowly? Most probably you have. But did you ever consider whether actually your database was at fault for the issue?

In our experience:
Database Performance = Application Performance.

In this eBook, we will walk you through the process of getting a 3x performance improvement on your Postgres database and 500x reduced data loaded from disk.

ABOUT

Database Performance = Application Performance

Often times, application performance is determined by the underlying database and its configuration – due to the fact that many applications and their ORMs (Object-Relational Mappings) are not aware of the SQL that's running and hiding behind an ORM call.

For example, in Ruby on Rails you might see something like this in the application code:

```
1   BackendWaitEvent.where(backend_id:
2   user.backends.first).pluck(:wait_event)
```

But only later realize that the SQL it produces is more like this:

```
1   SELECT  "backends".* FROM "backends" INNER JOIN "servers" ON
2   "backends"."server_id" = "servers"."id" INNER JOIN "organizations" ON
3   "servers"."organization_id" = "organizations"."organization_id" INNER
4   JOIN "organization_memberships" ON "organizations"."organization_id" =
5   "organization_memberships"."organization_id" WHERE
6   "organization_memberships"."user_id" = $1 AND
7   "organization_memberships"."accepted" = $2 ORDER BY
8   "backends"."backend_id" ASC LIMIT $3;
9   SELECT "backend_wait_events"."wait_event" FROM "backend_wait_events"
10  WHERE "backend_wait_events"."backend_id" = $1;
```

As we can see, these SQL statements have to do a bit of work to actually find the data we are looking for. To the application developer using the ORM however this looks like a simple function call that sometimes has high latency.

We can easily realize here:
Database Performance = Application Performance.

Missing Indices are the #1 database performance problem

In our experience, most development teams do not verify **all** new SQL that runs when they push a feature change.

Especially with ORMs in play, it is very difficult to know exactly which SQL statements get executed when reviewing a pull request that adds new functionality. Only when functionality is tested with realistic data on a staging system, or when you can see the effects of concurrent queries running on production, you truly learn what changed on the database side.

The most common mistake causing database performance issues is that developers forget to add an index – and often even if a feature is launched to production, you won't notice that missing index

for a few weeks or months, until the feature gets sufficient usage, or underlying data, that it becomes a performance bottleneck.

The most common mistake causing database performance issues is developers forgetting to add an index.

DID YOU KNOW?

Figuring out what's going on in your database(s)

Let's assume we want to find out whether you have any currently slow queries and missing indices on your PostgreSQL database. How could we go about this?

Reviewing Query Performance in PostgreSQL

One essential tool to achieve this is the `pg_stat_statements` extension in Postgres. It's bundled as part

of the contrib package, so you can install it easily to your database server – it may also already be enabled if you are using a managed database-as-a-service such as Heroku Postgres.

To check whether `pg_stat_statements` is enabled, and how to install it, you can follow this guide on our documentation.

Using pg_stat_statements to find expensive queries

Here is a standard query you can run on your database to get query statistics from `pg_stat_statements`:

```
SELECT queryid, calls, mean_time, substring(query for 100)
  FROM pg_stat_statements
 ORDER BY total_time DESC
 LIMIT 10;
```

This will then give us a list like the following, with the most expensive query on top (see next page):

```
   queryid   | calls  |    mean_time     |              substring
-------------+--------+------------------+-----------------------------------
  823659002  | 100856 |  212.20739523876 | SELECT “backend_wait_events” ...
 1908568318  |    224 | 392311.585268714 | COPY public.queries ...
 2996059654  |  59056 | 718.891097988979 | UPDATE “backends” ...
  107459272  |    223 | 189880.905045996 | COPY public.query_explains ...
 1819695266  |    223 |  119756.64852817 | COPY public.query_samples ...
 1615643520  |    224 |  90714.414896558 | COPY public.backend_wait_events ...
 3088208845  | 134836 | 87.2854475040199 | COPY “backend_wait_events” ...
  411003829  |   7103 |  983.00906357286 | UPDATE “backends” ...
  429818704  |    211 | 28399.7321560284 | COPY public.snapshot_benchmarks ...
 3773426307  |    224 | 19193.0874573839 | COPY public.backend_queries ...
(10 rows)
```

One thing to note is that `pg_stat_statements` records its statistics from the beginning of when it was installed, or alternatively, from when you've last reset the statistics.

Important: When using `pg_stat_statements` without a monitoring product like pganalyze, you can use the `pg_stat_statements_reset()` function to reset statistics.

Analyzing the performance of a specific slow query

Let's have a look at the above query output. How could we go about figuring out why the query starting with `SELECT „backend_wait_events“` is slow?

First of all, let's get the full query text as stored by `pg_stat_statements`, by querying just that `queryid`:

```
=> SELECT query FROM pg_stat_statements WHERE queryid = 823659002;

                                query
-------------------------------------------------------------------
 SELECT “backend_wait_events”.”wait_event” FROM “backend_wait_events”
 WHERE “backend_wait_events”.”backend_id” = $1
(1 row)
```

Here we can see, `pg_stat_statements` records not a specific invocation of the query, but rather an aggregated, normalized form of the query. Similar queries are grouped together based on the `queryid`, and the text gets normalized, so if you had `“backend_id = ‘something’”` in the original SQL, it stores `“backend_id = $1”` instead.

This is mostly for the user's benefit, but has the downside that we can't run `EXPLAIN` on the query text:

```
1   => EXPLAIN SELECT "backend_wait_events"."wait_event" FROM backend_wait_events
2   WHERE backend_id = $1;
3   ERROR:  there is no parameter $1
4   LINE 1: ..."wait_event" FROM backend_wait_events WHERE backend_id = $1;
```

This makes sense of course, since **Postgres execution plans are dependent on the specific values you are querying for** - we need to know the value of $1 in order to run `EXPLAIN`.

EXPLAIN lets you determine the execution plan for a query by showing how Postgres executes it, e.g. by letting you know whether its going for an Index Scan (typically good) or a Sequential Scan (often slow, except on very small tables).

POSTGRES EXPLAIN

Now, we could just happen to know the value of `backend_id` and replace this ourselves, allowing us to run the `EXPLAIN`:

```
=> EXPLAIN SELECT "backend_wait_events"."wait_event" FROM backend_wait_events
WHERE backend_id = 'd95d627c-bea7-4c7e-bea5-4f69e18fe53a';
                                QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on backend_wait_events  (cost=0.00..168374.85 rows=268 width=14)
   Filter: (backend_id = 'd95d627c-bea7-4c7e-bea5-4f69e18fe53a'::uuid)
 JIT:
   Functions: 4
   Inlining: false
   Optimization: false
(6 rows)
```

But often, we don't know the values for these parameters, leading us to the next question: **How can we determine the bind parameter values for queries in pg_stat_statements?**

Finding bind parameter values for slow queries

In order to get the full query text, we have two choices: First, we can utilize Postgres' `pg_stat_activity` table, which shows the currently running queries. If you don't use parameters in your own application code (i.e. you send all values in the query text itself), this will work, but requires some extra effort by sampling that table frequently.

As a more generic approach that works with both the bind parameter values for `pg_stat_statements`, and those sent separately by the application, we

turn to the Postgres logging system.

Understanding the Postgres Logging System

Postgres generates a large amount of log events, and it takes a lot of effort to review and parse log files. For this specific example however, we are just looking at a single log event, the slow query log output, controlled by `log_min_duration_statement`.

log_min_duration_statement vs log_statement:
For those familiar with Postgres config options, you may wonder why we are recommending the use of `log_min_duration_statement` instead of `log_statement`. Whilst you could utilize `log_statement = all` to get the full query text for every single statement that has run, this very rarely makes sense in production as it might take down your production system, due to the overhead for log output on very fast queries. We therefore recommend only using `log_min_duration_statement` on production systems.

LOGS

We can set `log_min_duration_statement` to a specific threshold, and any SQL queries running longer than that duration will have the full query text logged to the Postgres log files. Typically, it makes sense to start with a threshold like 1000 ms, and lower

that slightly if needed as **the goal here is to not log every query, but rather find the specific query text for outlier queries**.

Once enabled, the output looks like this:

```
LOG:  duration: 454.746 ms  execute a8: SELECT “backend_wait_events”.”wait_event”
FROM “backend_wait_events” WHERE “backend_wait_events”.”backend_id” = $1
DETAIL:  parameters: $1 = ‘6d2d2787-6c27-4d81-807f-37989dc6b9b0’
```

As you can see we can get the parameters as sent by the client, and if `pg_stat_statements` would replace any values, those would also be correctly reflected in the log event. We can now run `EXPLAIN` on this, yielding the correct query plan:

```
=> EXPLAIN SELECT “backend_wait_events”.”wait_event” FROM “backend_wait_events”
WHERE “backend_wait_events”.”backend_id” = ‘6d2d2787-6c27-4d81-807f-37989dc6b9b0’;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on backend_wait_events  (cost=0.00..168374.85 rows=30012 width=14)
   Filter: (backend_id = ‘6d2d2787-6c27-4d81-807f-37989dc6b9b0’::uuid)
 JIT:
   Functions: 4
   Inlining: false
   Optimization: false
(6 rows)
```

Gathering EXPLAIN plans automatically using auto_explain

The above process works for running a few `EXPLAIN`s here and there, but it's too much effort to run systematically. In addition, **if you only look at the log files a day or two later, you might get a different execution plan than what had actually occurred when the slow query happened.**

We therefore turn to another very useful Postgres extension: auto_explain.

`auto_explain` is also bundled with Postgres in the contrib package, like `pg_stat_statements`, and has to be enabled on your database. See our setup guide on our documentation. Once enabled, the `auto_explain.log_min_duration` setting is determining which queries get their `EXPLAIN` plan logged. To start, we recommend setting this to 1000 ms, and lowering it as needed.

You will then get plans like this into your log file, as slow queries happen (see below):

```
LOG:  duration: 454.730 ms  plan:
      Query Text: SELECT "backend_wait_events"."wait_event" FROM "backend_wait_
events" WHERE "backend_wait_events"."backend_id" = $1
      Seq Scan on public.backend_wait_events  (cost=0.00..168374.85 rows=30012
width=14) (actual rows=32343 loops=1)
        Output: wait_event
        Filter: (backend_wait_events.backend_id = '6d2d2787-6c27-4d81-807f-
37989dc6b9b0'::uuid)
        Rows Removed by Filter: 6445165
        Buffers: shared hit=16145 read=71261
```


Determining missing indices based on EXPLAIN plans

Now, lets review the EXPLAIN plan we had earlier, and let's try to understand how we could improve performance. We can run the `EXPLAIN` with the

`ANALYZE` and `BUFFERS` options, for full details on the query execution:

```
=> EXPLAIN (ANALYZE, BUFFERS) SELECT "backend_wait_events"."wait_event" FROM
"backend_wait_events" WHERE "backend_wait_events"."backend_id" = '6d2d2787-6c27-
4d81-807f-37989dc6b9b0';
                                      QUERY PLAN
------------------------------------------------------------------------------
 Seq Scan on backend_wait_events  (cost=0.00..168374.85 rows=30012 width=14)
 (actual time=3.004..537.623 rows=32343 loops=1)
   Filter: (backend_id = '6d2d2787-6c27-4d81-807f-37989dc6b9b0'::uuid)
   Rows Removed by Filter: 6445165
   Buffers: shared hit=417 read=86989
 Planning Time: 0.100 ms
 JIT:
   Functions: 4
   Generation Time: 0.361 ms
   Inlining: false
   Inlining Time: 0.000 ms
   Optimization: false
   Optimization Time: 0.262 ms
   Emission Time: 2.210 ms
 Execution Time: 628.484 ms
(14 rows)
```

First of all, you can see **JIT** referenced here, which is a recent addition to PostgreSQL, available on Postgres 11 or newer. It got activated here since the query is quite expensive to run and processes a lot of rows. If you want to learn more about JIT, check out our blog post about it.

When reading an `EXPLAIN` plan it makes sense to focus on the most expensive part of the plan. Here the plan is simple, since we only have a single plan node – the `Seq Scan` node. Sequential scans read through the table data sequentially (hence the

name), without using any index.

You can see that Postgres is filtering the scan with the specific `backend_id` it is looking for, so it has to throw away a lot of rows, as indicated by `Rows Removed by Filter:`. Postgres is also loading a lot of data, as indicated by the `Buffers:` information – specifically, it is loading 680 MB of data from disk (86989 buffers read, multiplied by the default Postgres block size of 8 KB).

Now, the next step would be to understand why Postgres is doing the sequential scan – maybe there is no index?

The simplest method to check this with standard tools is to simply look at the table in the Postgres client, psql, and use the `\d` command:

```
=> \d backend_wait_events
                                Table “public.backend_wait_events”
        Column         |    Type   | Nullable |      Default
-----------------------+-----------+----------+-------------------
 backend_wait_event_id | uuid      | not null | gen_random_uuid()
 server_id             | uuid      | not null |
 backend_id            | uuid      | not null |
 seen_at               | timestamp | not null |
 wait_event_type       | text      | not null |
 wait_event            | text      | not null |
Indexes:
    “backend_wait_events_pkey” PRIMARY KEY, btree (backend_wait_event_id)
```

We can see that there is a single index on the table, on the primary key. There is no index on the field we are querying for, and therefore a sequential scan was necessary.

Now, let's say we create an index like this:

```
CREATE INDEX CONCURRENTLY ON backend_wait_events(backend_id);
```

And then re-run the `EXPLAIN`:

```
=> EXPLAIN (ANALYZE, BUFFERS) SELECT "backend_wait_events"."wait_event" FROM
"backend_wait_events" WHERE "backend_wait_events"."backend_id" = '6d2d2787-6c27-
4d81-807f-37989dc6b9b0';
                                                                                QUERY
PLAN
-------------------------------------------------------------------------------------------
-----------------------------------------------------
Bitmap Heap Scan on backend_wait_events  (cost=697.03..61932.29 rows=30012
width=14) (actual time=9.044..197.026 rows=32343 loops=1)
   Recheck Cond: (backend_id = '6d2d2787-6c27-4d81-807f-37989dc6b9b0'::uuid)
   Heap Blocks: exact=26451
   Buffers: shared hit=126 read=26452 written=6
   ->  Bitmap Index Scan on backend_wait_events_backend_id_idx
(cost=0.00..689.52 rows=30012 width=0) (actual time=5.537..5.539 rows=32343
loops=1)
         Index Cond: (backend_id = '6d2d2787-6c27-4d81-807f-37989dc6b9b0'::uuid)
         Buffers: shared hit=126 read=1
 Planning Time: 0.154 ms
 Execution Time: 286.110 ms
(9 rows)
```

We are now using a `Bitmap Index Scan` instead of a sequential scan. **We can see that performance improved 2x based on that index. Very**

good, but can we do better?

In fact we can! As we see in the new plan we are still loading 26451 blocks (207 MB) from the table itself, in order to get the value of the `wait_event` column we are looking for. **What if we simply included that column in the index?**

In older Postgres versions, you can create a multi-column index like this:

```
1   CREATE INDEX CONCURRENTLY ON backend_wait_events(backend_id, wait_event);
```

But, since we are testing on Postgres 11 here, we can also use the new `INCLUDE` keyword to specify non-key columns we want to have present in the index:

```
1   CREATE INDEX CONCURRENTLY ON backend_wait_events(backend_id) INCLUDE (wait_event);
```

The new plan now looks like this (see below):

```
=> EXPLAIN (ANALYZE, BUFFERS) SELECT "backend_wait_events"."wait_event" FROM
"backend_wait_events" WHERE "backend_wait_events"."backend_id" = '6d2d2787-6c27-
4d81-807f-37989dc6b9b0';
                                QUERY PLAN
---------------------------------------------------------------------------
 Index Only Scan using backend_wait_events_backend_id_wait_event_idx
on backend_wait_events  (cost=0.43..1496.33 rows=35194 width=14) (actual
time=0.017..96.079 rows=32343 loops=1)
   Index Cond: (backend_id = '6d2d2787-6c27-4d81-807f-37989dc6b9b0'::uuid)
   Heap Fetches: 0
   Buffers: shared read=168
 Planning Time: 0.059 ms
 Execution Time: 188.495 ms
(6 rows)
```

That yielded another **1.5x performance improvement**. In addition, we reduced the amount of data loaded from disk to 1.3 MB, **a 500x difference to the initial plan**! The reduction in data being loaded will reduce stress on the disk, and allow other queries to use the I/O bandwidth that is now freed up.

We can see that **it pays off to optimize query performance**. However it can be a lot of work to run all these queries and work through the data for every query. This is one of the main reasons we are building pganalyze.

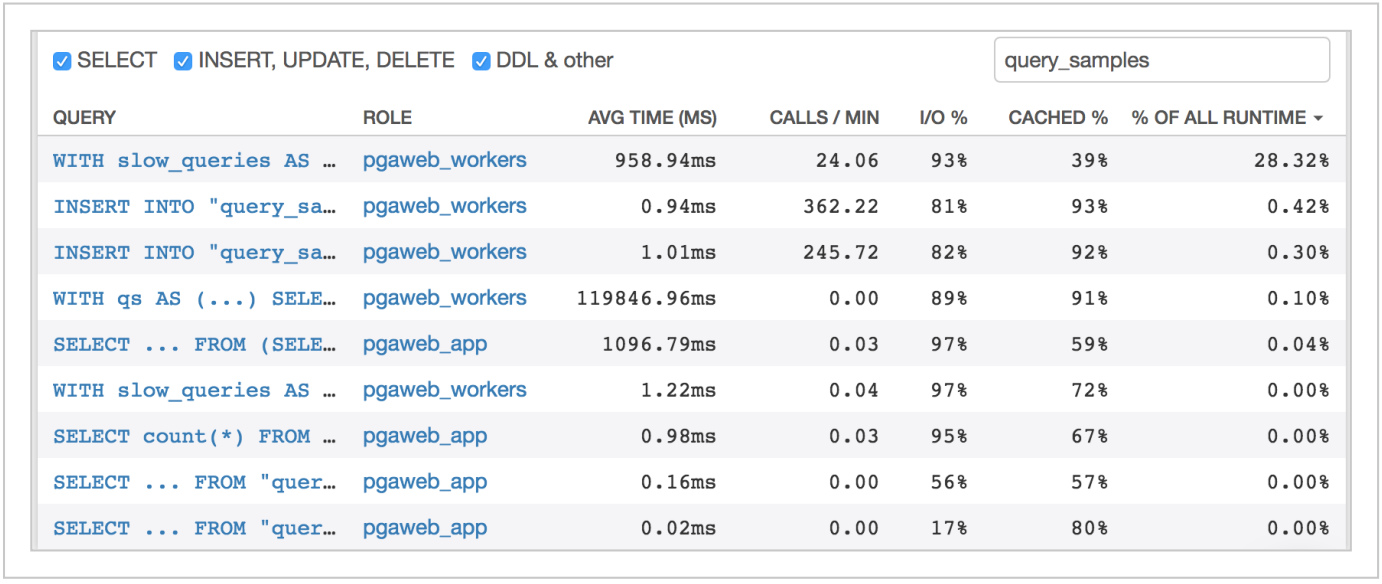
With pganalyze, we automate this process for you, so you quickly find the root cause for slow queries, and add the correct indices in no time.

pganalyze: Query information from statistics tables and your log files in one place

pganalyze was built with both DBAs and application developers in mind. We automate processes for you that are usually time-intensive, and not accessible to the broader development team.

Identify most expensive queries with the top-down query view

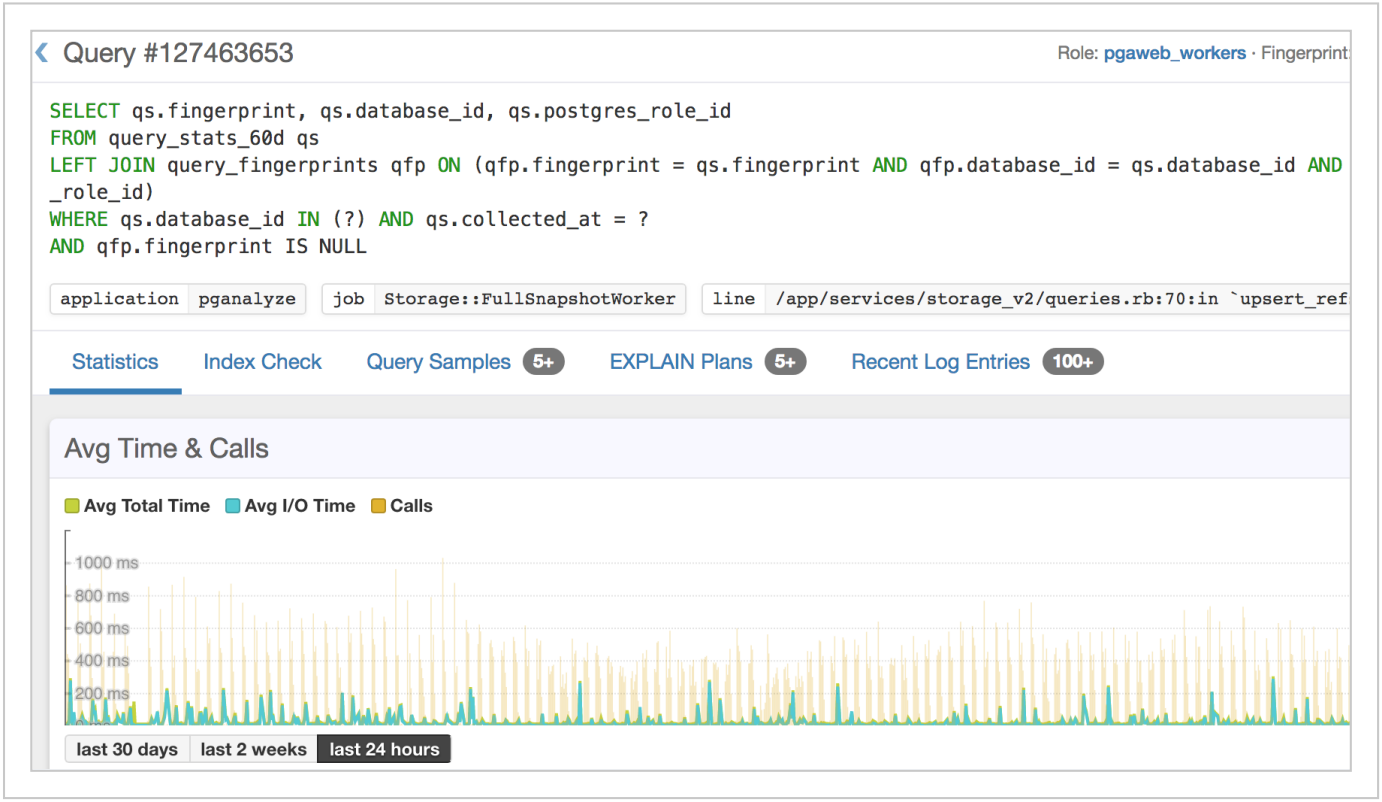
Firstly, we provide you with the top-down view of all queries that are active in your database, so you can quickly see the most expensive query for a given timeframe:



QUERY	ROLE	AVG TIME (MS)	CALLS / MIN	I/O %	CACHED %	% OF ALL RUNTIME
WITH slow_queries AS …	pgaweb_workers	958.94ms	24.06	93%	39%	28.32%
INSERT INTO "query_sa…	pgaweb_workers	0.94ms	362.22	81%	93%	0.42%
INSERT INTO "query_sa…	pgaweb_workers	1.01ms	245.72	82%	92%	0.30%
WITH qs AS (...) SELE…	pgaweb_workers	119846.96ms	0.00	89%	91%	0.10%
SELECT ... FROM (SELE…	pgaweb_app	1096.79ms	0.03	97%	59%	0.04%
WITH slow_queries AS …	pgaweb_workers	1.22ms	0.04	97%	72%	0.00%
SELECT count(*) FROM …	pgaweb_app	0.98ms	0.03	95%	67%	0.00%
SELECT ... FROM "quer…	pgaweb_app	0.16ms	0.00	56%	57%	0.00%
SELECT ... FROM "quer…	pgaweb_app	0.02ms	0.00	17%	80%	0.00%

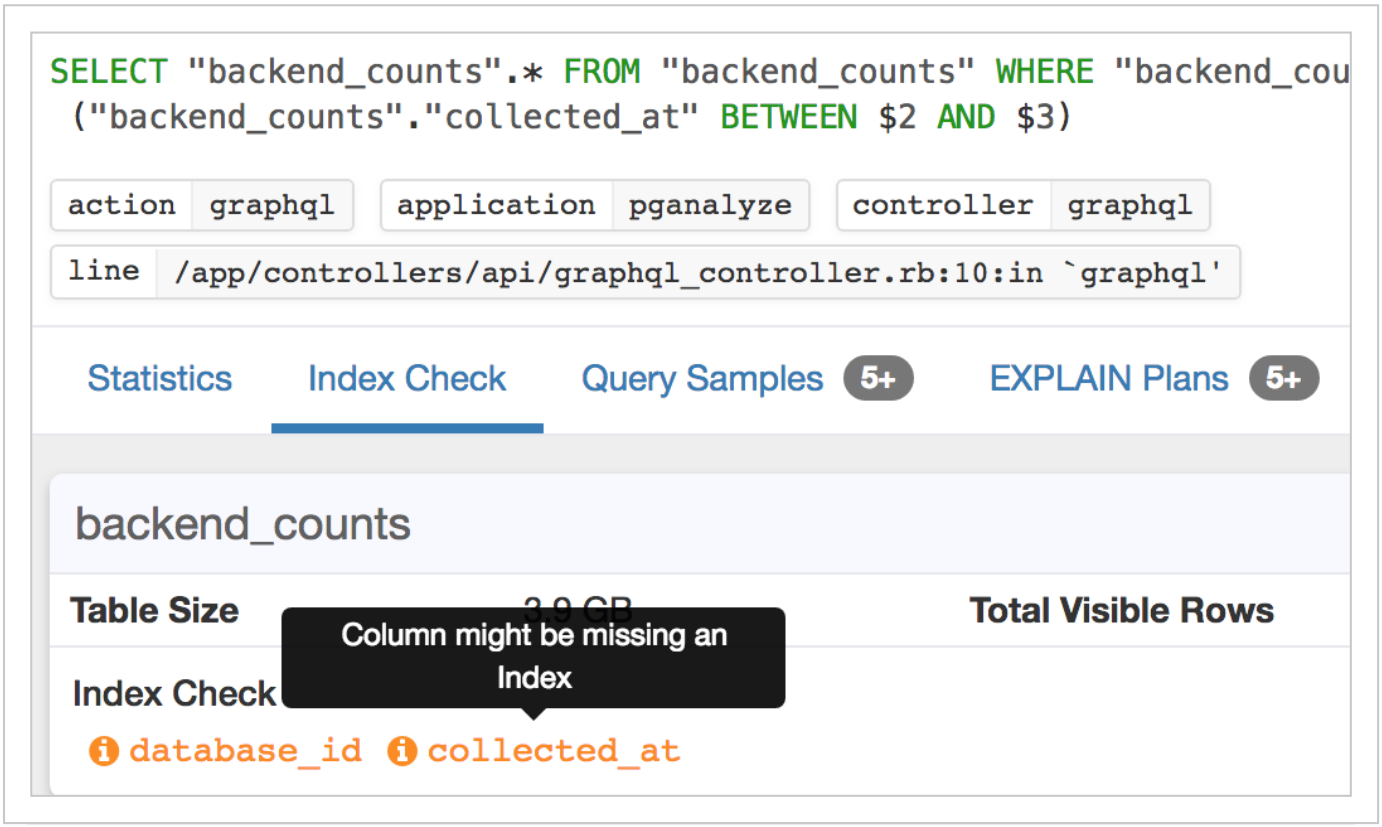
See query statistics over time

Secondly, we have detailed pages for each query, providing statistics over time:



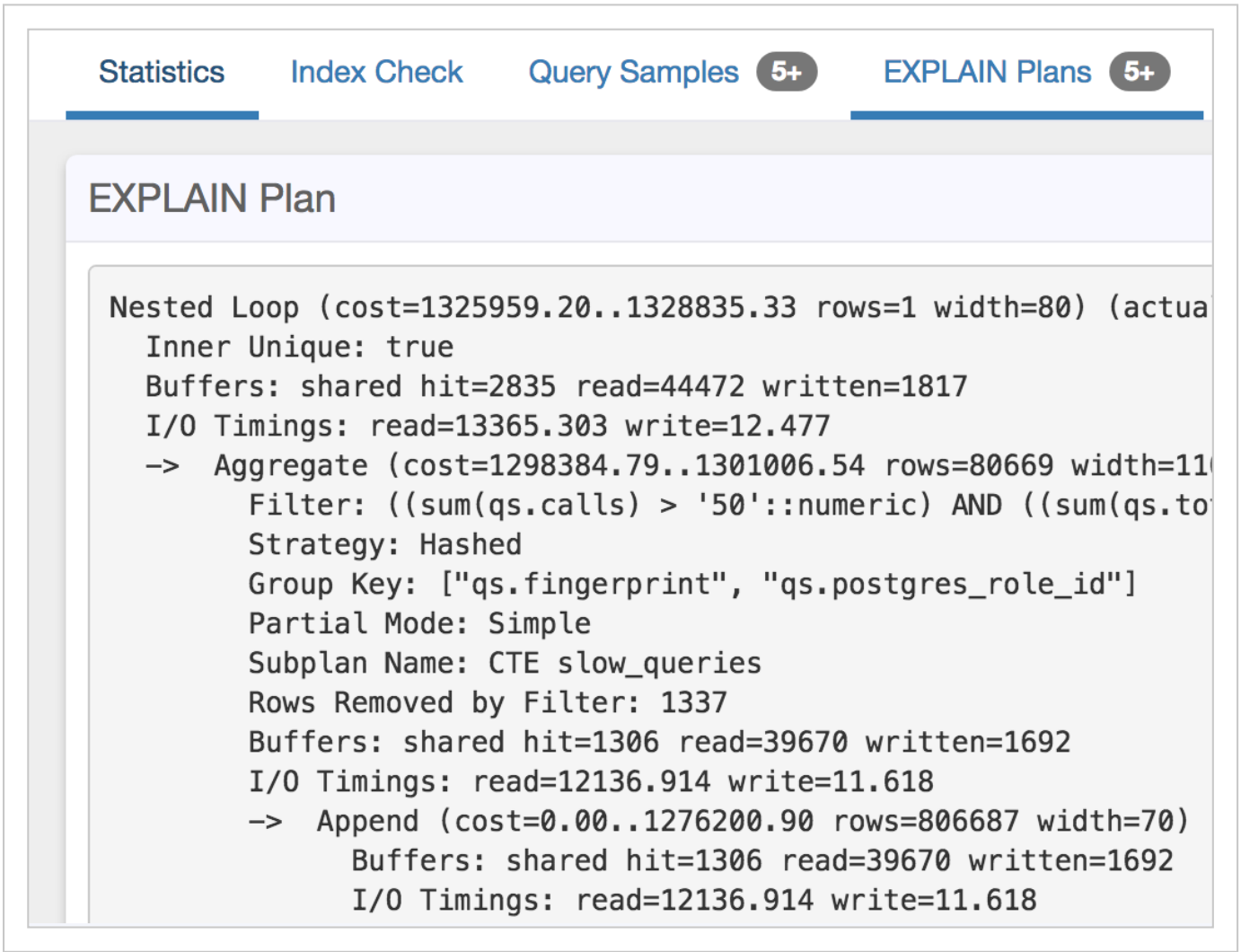
Find missing indices with the pganalyze Index Check

Using the pganalyze Index Check you can quickly find missing indices for a query:



Understand your logs

We integrate with the Postgres logging system, and automatically collect log events for you using pganalyze Log Insights - supported for on-premise and major cloud providers like Amazon RDS. Log Insights associates log events to queries and other database objects. When you enable `auto_explain`, you will automatically see `EXPLAIN` plans on the query details page:



There are many more pganalyze features, such as VACUUM monitoring, Connection Tracing and sophisticated role management on pganalyze which makes it easy for DBAs and developers to identify the root cause of performance issues, optimize queries and to get alerts about critical issues.

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!