Postgres on AWS
What's New &
How to Make the Most of It

Our Agenda today:

RDS vs Aurora vs CNPG

Let's talk about I/O

Aurora Tiered Caching

Aurora Serverless v2 Platform Version 3

Extensibility

Debuggability & Community Support

Monitoring

pganalyze vs Database Insights

Database Savings Plans

RDS vs **Aurora** vs **CNPG**

Documentation > Amazon RDS > User Guide

What is Amazon Relational Database Service (Amazon RDS)?

PDF RSS Focus mode

Amazon Relational Database Service (Amazon RDS) is a web service that makes it easier to set up, operate, and scale a relational database in the AWS Cloud. It provides cost-efficient, resizable capacity for an industry-standard relational database and manages common database administration tasks.

Note

This guide covers Amazon RDS database engines other than Amazon Aurora. For information about using Amazon Aurora, see the *Amazon Aurora User Guide*.

If you are new to AWS products and services, begin learning more with the following resources:

- For an overview of all AWS products, see What is cloud computing?
- Amazon Web Services provides a number of database services. To learn more about the variety of database options available on AWS, see Choosing an AWS database service and Running databases on AWS.

C

R

For the purpose of this webinar, RDS means not Aurora

CloudNativePG

Blog Tutorials Releases Documentation Support End Users

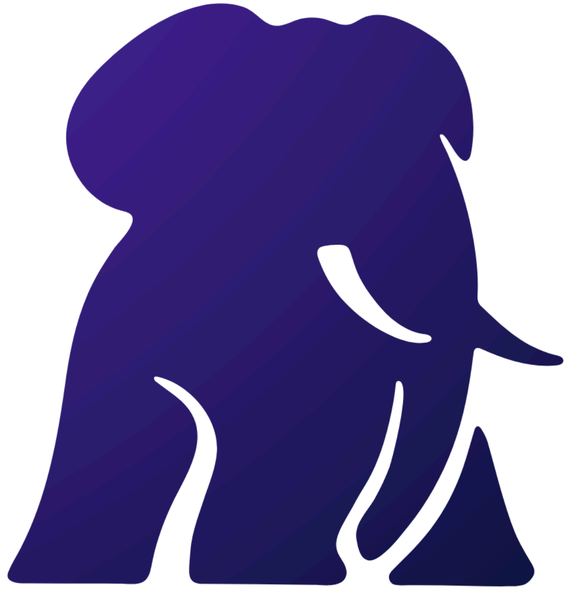
Star 7,545

Run PostgreSQL.

The Kubernetes way.

CloudNativePG is the Kubernetes operator that covers the full lifecycle of a highly available PostgreSQL database cluster with a primary/ standby architecture, using native streaming replication.

View on GitHub



CloudNativePG (CNPG) has become the Postgres operator of choice in Kubernetes

pganalyze

Pricing can be complicated, but generally: CNPG < RDS < Aurora

▼ RDS Price Calculations

```
Storage and IOPS
1TB = 1,000GB
1,000GB x $0.125 per GB-month = $125 storage
5,000 IOPS x $0.10 per IOPS-month = $500 IOPS
Instance Hours
730 hours x $1.92 price per hour = $1,401.60 instance hours
Total
$125 storage + $500 IOPS + $1,401.60 instance hours = $2,026.60
```

▼ Aurora Price Calculations

```
Storage and IOPS
1TB = 1,000GB
1,000GB x $0.225 per GB-month = $225 storage
Instance Hours
730 hours x $3.02 price per hour = $2,204.60 instance hours
Total
$225 storage + $2,204.60 instance hours = $2,429.60
```

In this scenario, RDS is 16.59% less than Aurora, making RDS the cost-effective choice. If, say, the e-commerce platform needed to increase the provisioned IOPS, Aurora would become the cost-effective choice after the breakeven point 9,030 IOPS, since I/O is included in the price of Aurora I/O-Optimized but charged $0.10 per IOPS-month for RDS Provisioned IOPS (io3).

Source

Let's talk about I/O

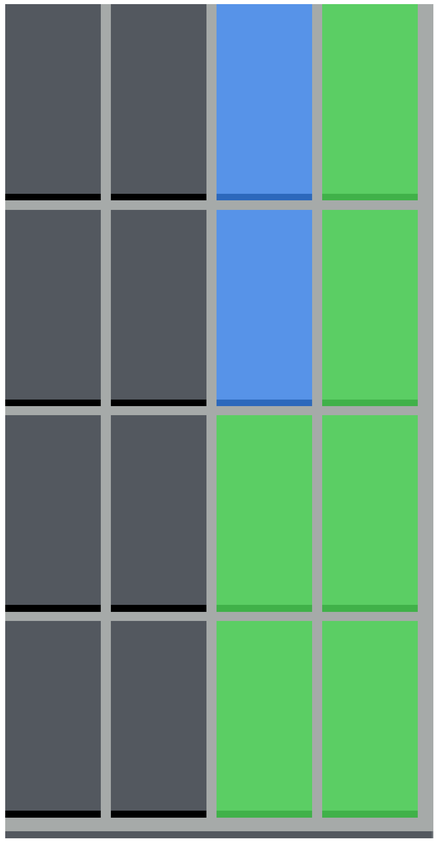
The Two Caches That Matter

pganalyze

* except on Aurora

Postgres Shared Buffers

8kB Page



e.g. 128kB shared_buffers

Pinned Buffer
Actively being read/used by a backend

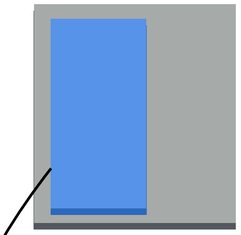
Dirty Buffer
Previously written to, not yet persisted to disk

Unpinned Buffer
Unused buffer, may contain cached data

pganalyze

shared_buffers are not (just) a cache, they are essential for writes.

1. Remember changed page in shared_buffers



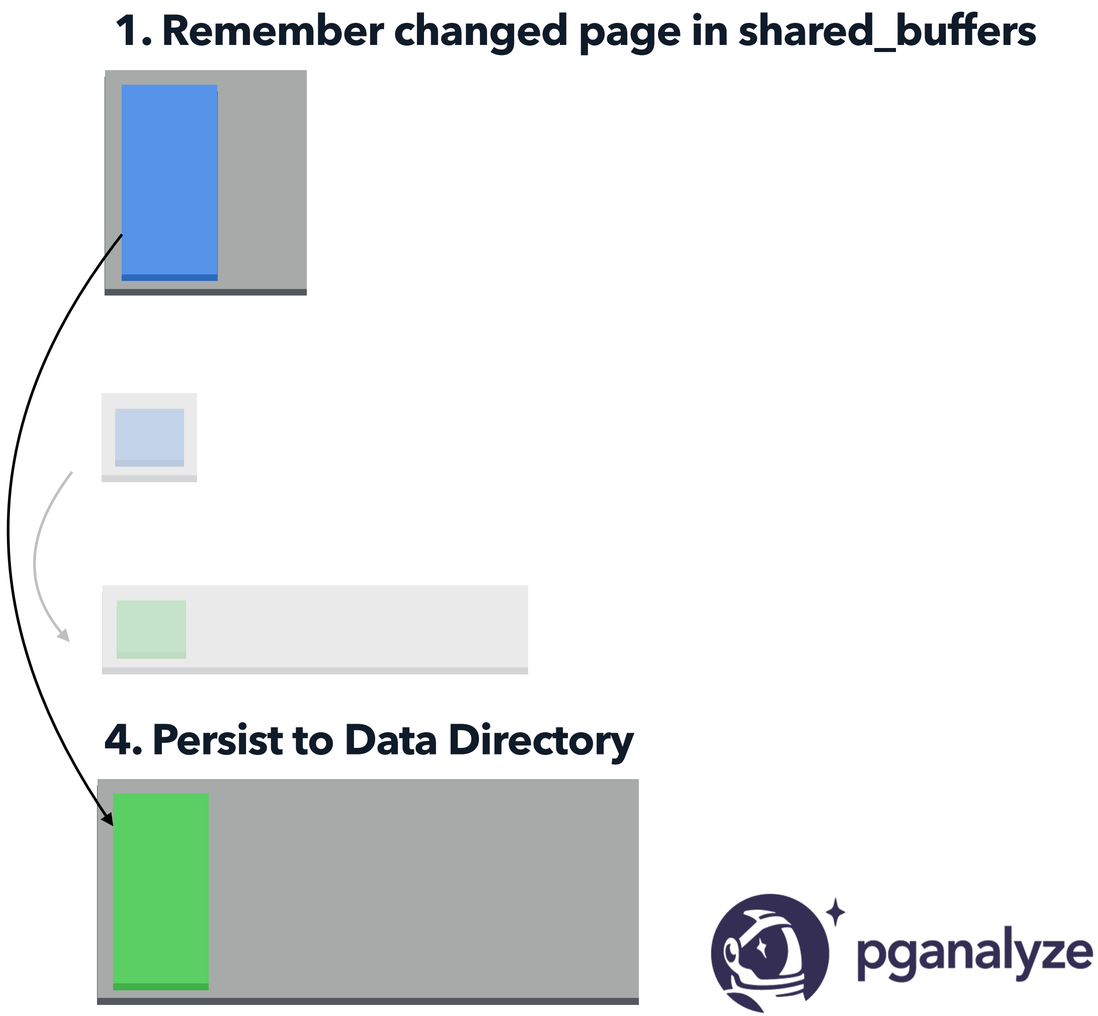
2. Remember changed rows (or full page) in wal_buffers

3. Persist to WAL

4. Persist to Data Directory

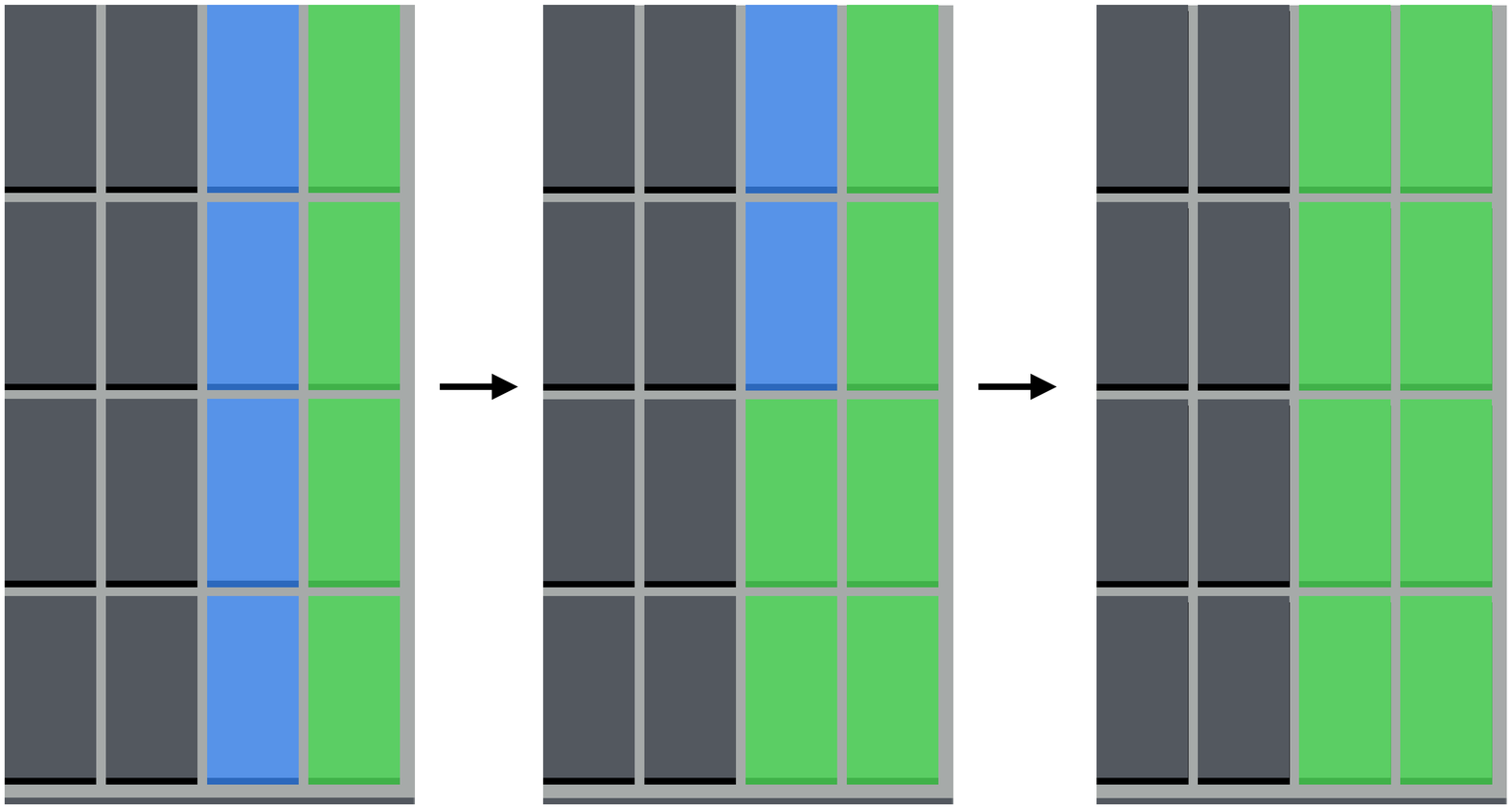
pganalyze

shared_buffers are not (just) a cache, they are essential for writes.



pganalyze

Background Writer



pganalyze

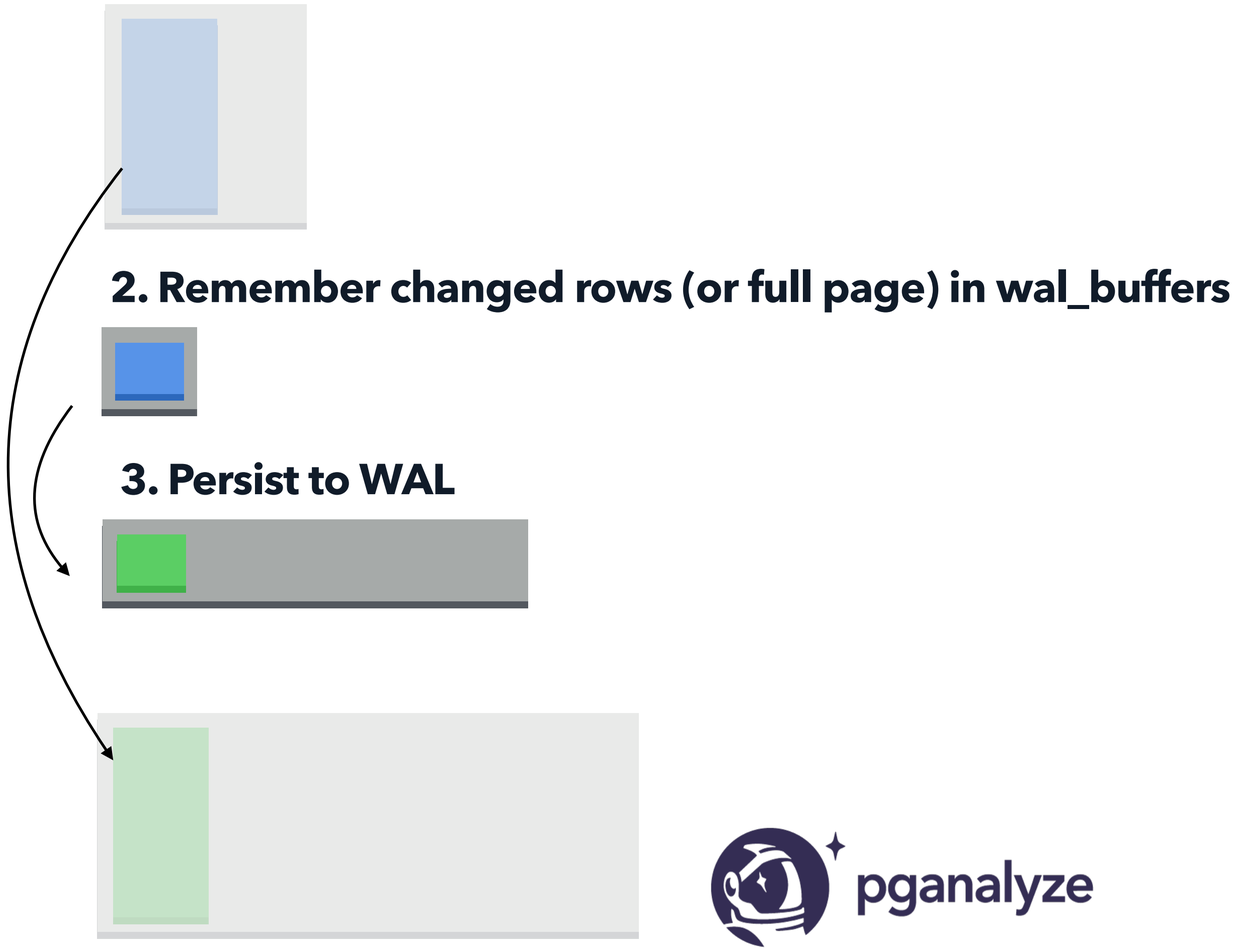
Background Writer

Tune to run more often for busy workloads
=> **reduce bgwriter_delay**

If background worker doesn't do its job in time,
Individual queries might write dirty buffers before they can read.
(pg_stat_statements.shared_blks_written)

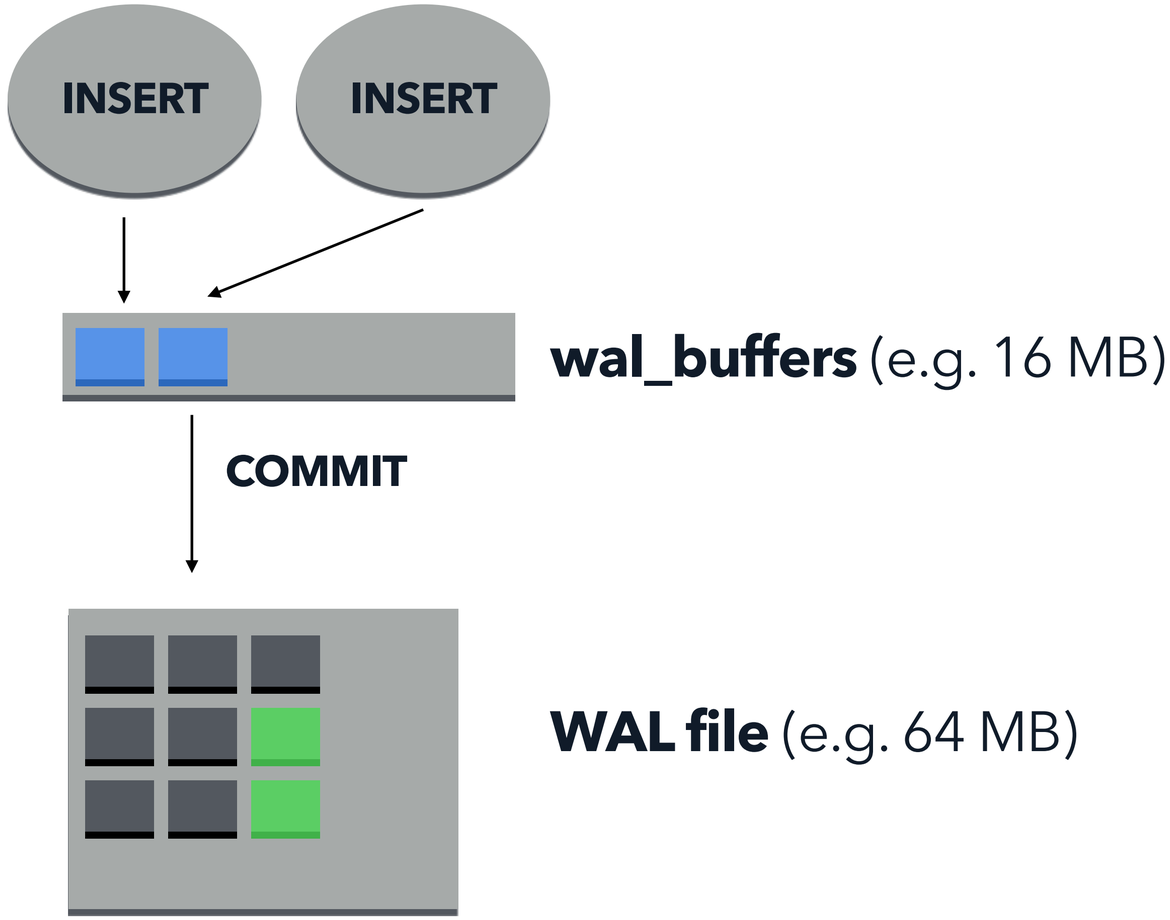
But: If it runs too often, you'll create additional I/O.
(a dirty page can be "re-used" for a write within the same checkpoint)

Persisting Changes to the WAL



pganalyze

wal_buffers



wal_buffers (e.g. 16 MB)

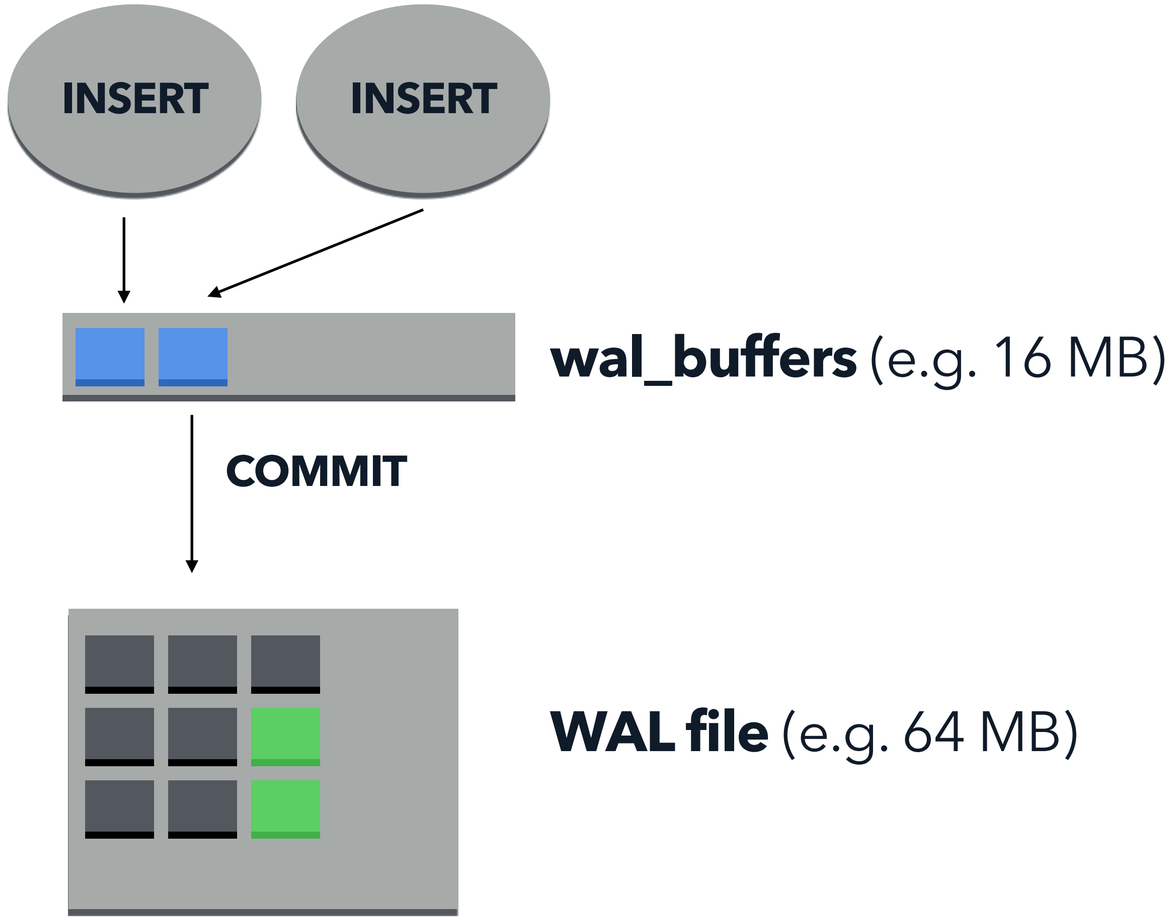
WAL file (e.g. 64 MB)

pganalyze

synchronous_commit = off

(sometimes)

synchronous_commit = on

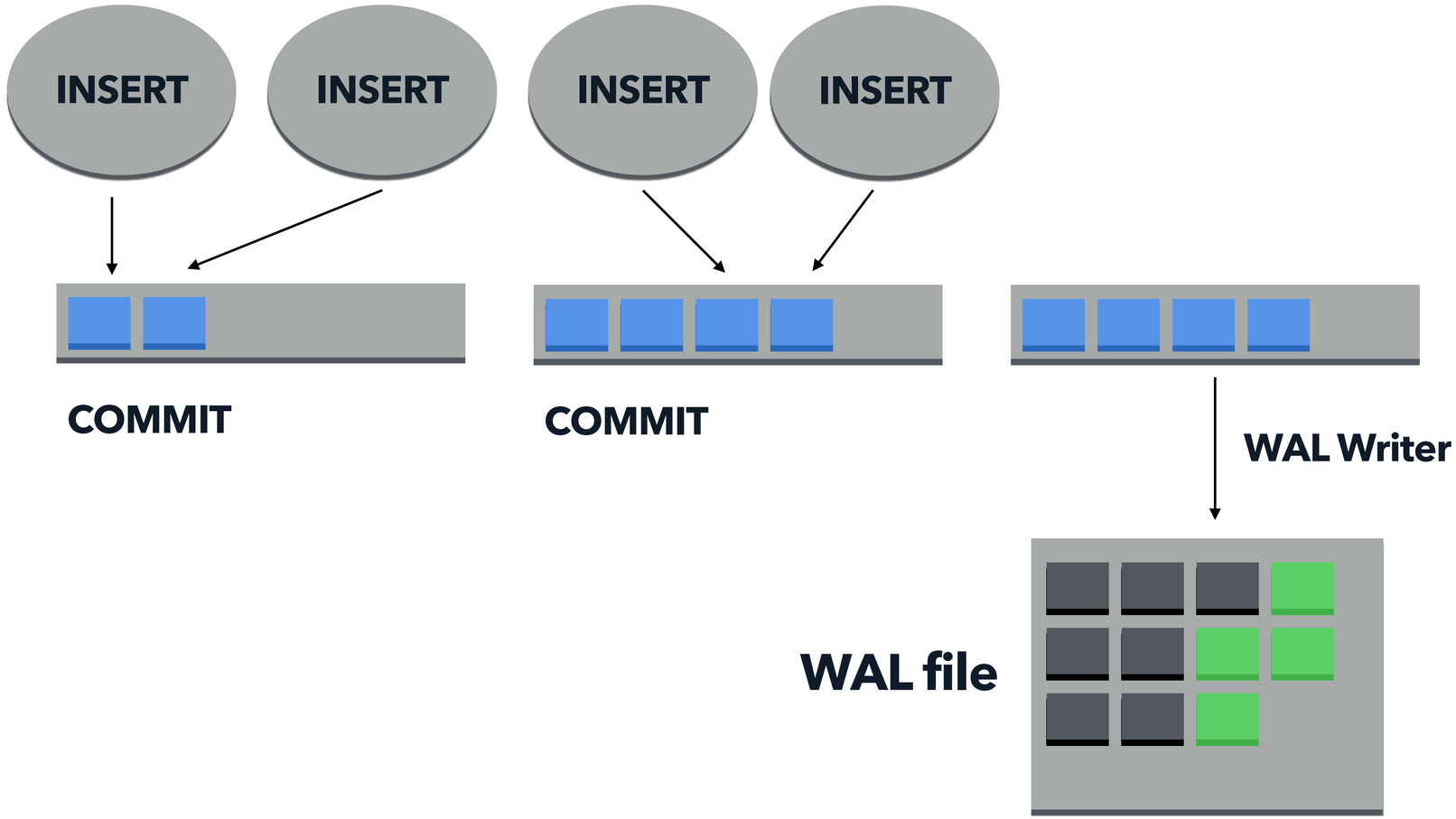


wal_buffers (e.g. 16 MB)

WAL file (e.g. 64 MB)

pganalyze

synchronous_commit = off



pganalyze

synchronous_commit = off

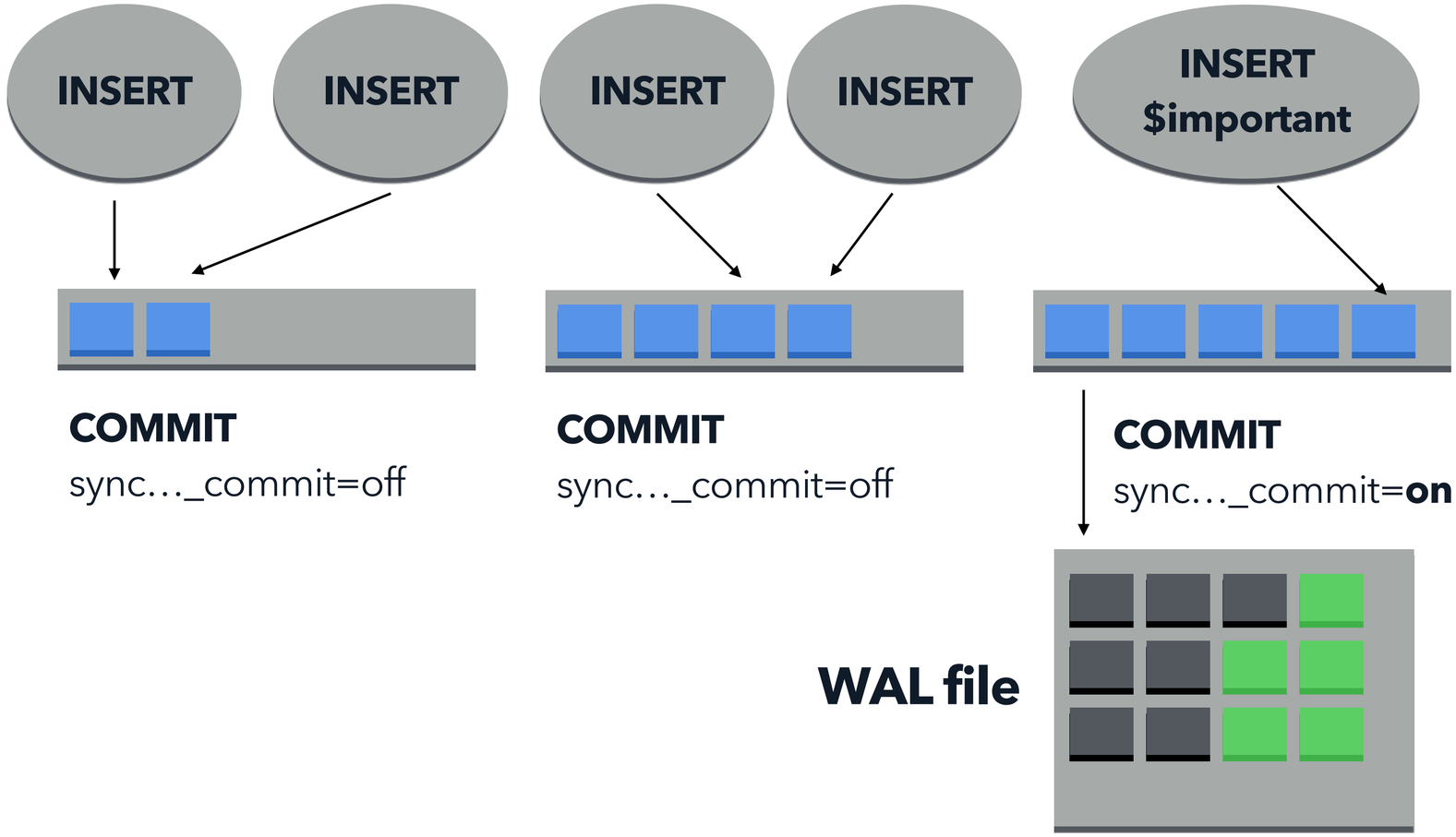
! You may loose data not yet persisted to WAL in a crash.

The database will be consistent
(just missing that most recent data),
no risk of corruption.

“This parameter **can be changed at any time**; the behavior for any one transaction is determined by the setting in effect when it commits.

It is therefore possible, and useful, to **have some transactions commit synchronously and others asynchronously**.”

synchronous_commit = [SET per transaction]



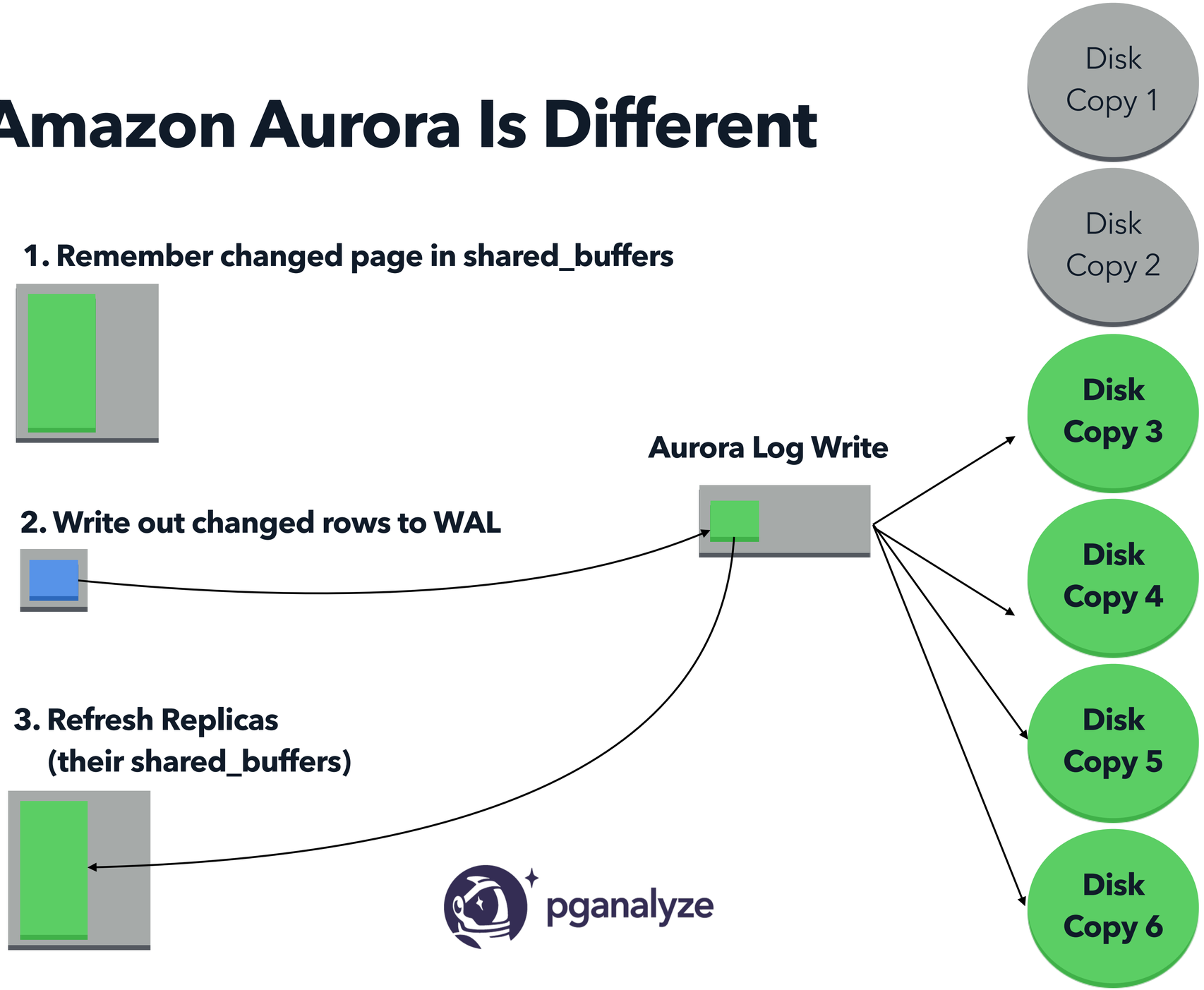
pganalyze

If you make heavy use of
synchronous_commit = off ...

- Consider lowering **wal_writer_delay**
  (to write WAL more frequently, avoiding flushes during individual commits)

- Increase **wal_buffers** to a multiple of wal_segment_size

Amazon Aurora Is Different



Amazon Aurora Is Different

No Full Page Writes

“No Checkpoints”
(Heavily Modified Checkpointer
+ Background Writer)

Aurora Is Not Always Better

Both Read and Write IOPS are charged extra

Read I/Os are always charged per 8kB disk page

Read I/Os will be slower (sometimes)

Write I/Os are always charged per 4kB log record

Write I/O with synchronous_commit=on will be slower

“An **unoptimized SQL query** can incur higher I/Os as compared to an optimized query, because it needs to scan a lot of pages to get to the final query result.

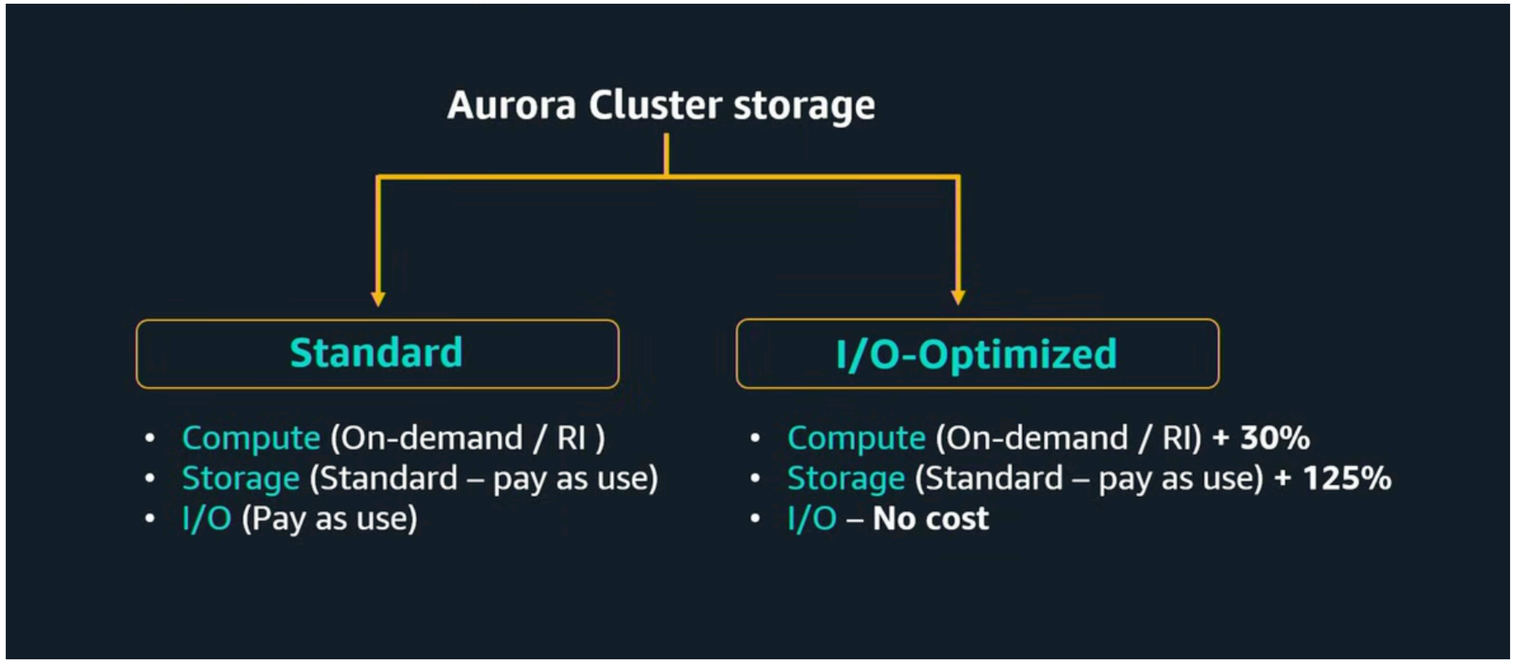
Typically, **this is the most common cause for higher Aurora I/Os.**”

Amazon Aurora I/O Cost Optimization Methodology

Aurora I/O Optimized changes this equation

"Aurora I/O-Optimized is the best choice
when your I/O spending is 25% or more
of your total Aurora database spending."

Aurora I/O Optimized changes this equation



AWS Re:invent 2025

pganalyze

Aurora Tiered Caching

This is not new, but its not well known:

AWS Database Blog

New – Amazon Aurora Optimized Reads for Aurora PostgreSQL with up to 8x query latency improvement for I/O-intensive applications

by Gowri Balasubramanian, Peipei Yin, and Jeremy Schneider | on 08 NOV 2023 | in Advanced (300), Amazon Aurora | Permalink | Comments | Share

Amazon Aurora is a MySQL- and PostgreSQL-compatible relational database built for the cloud. Aurora combines the performance and availability of traditional enterprise databases with the simplicity and cost-effectiveness of open-source databases. We are excited to announce the launch of the Optimized Reads feature for Aurora PostgreSQL. Aurora Optimized Reads delivers up to 8x improved query latency and up to 30% cost savings compared to instances without it, for applications with large datasets that exceed the memory capacity of a database instance. This new price-performance feature is available on AWS Graviton-based db.r6gd and Intel-based db.r6id instances that support non-volatile memory express (NVMe) storage.

Aurora Optimized Reads supports two capabilities:

- **Tiered cache** – This allows you to extend your DB instance caching capacity by utilizing the local NVMe storage. It automatically caches database pages about to be evicted from the in-memory database buffer pool, offering up to eight times better latency for queries that were previously fetching data from Aurora storage.
- **Temporary objects** – These are hosted on local NVMe storage instead of Amazon Elastic Block Store (Amazon EBS) based storage. This enables better latency and throughput for queries that sort, join, or merge large volumes of data that don't fit within the memory configured for those operations.

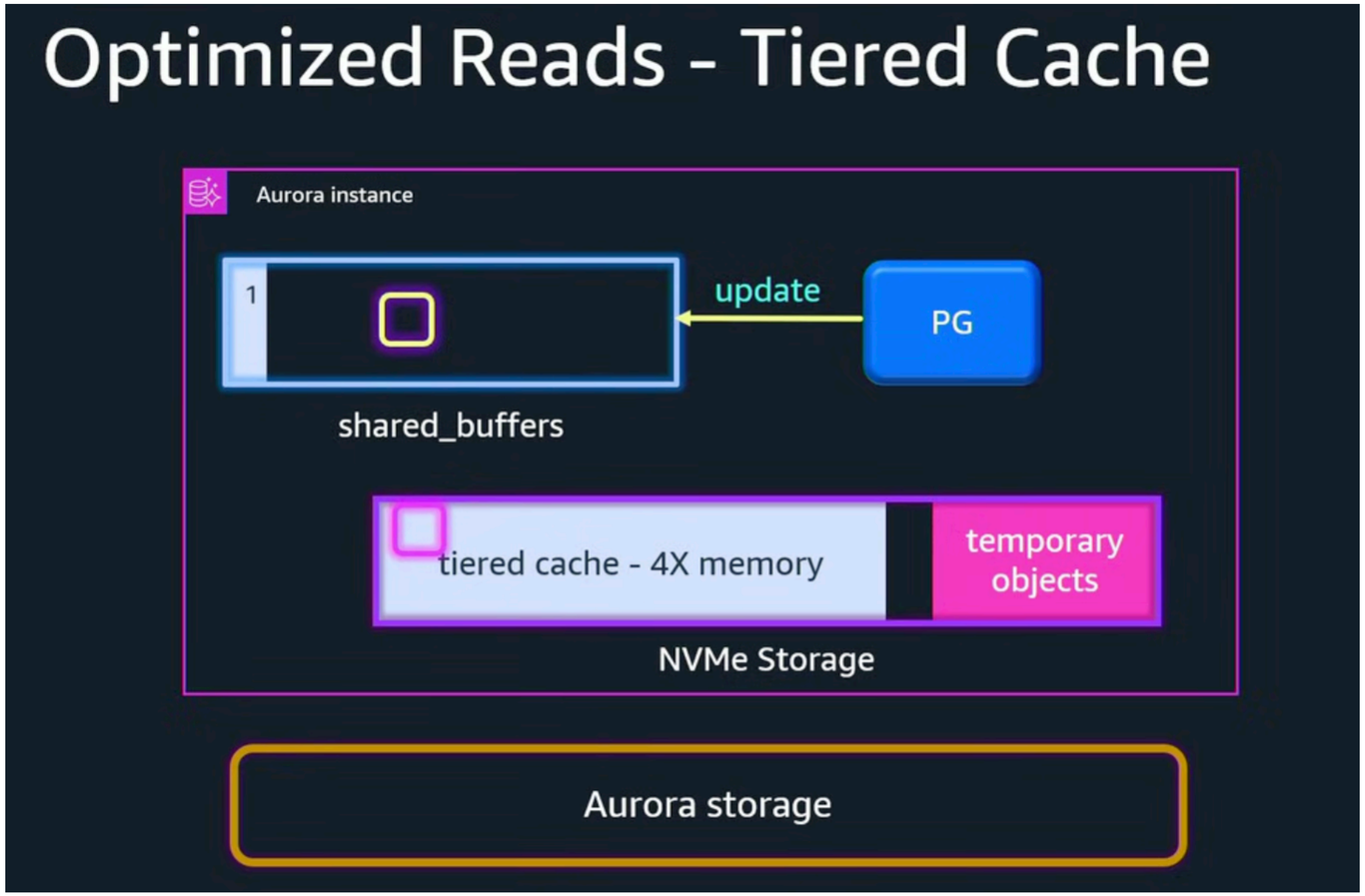
In this post, we discuss the Optimized Reads feature, typical use cases, and feature availability by engine and storage configuration. We dive deep into the tiered cache capability and how it can improve the query performance of latency-sensitive workloads and monitoring options. We also provide an overview of the temporary objects capability.

Resources

Getting Started
What's New

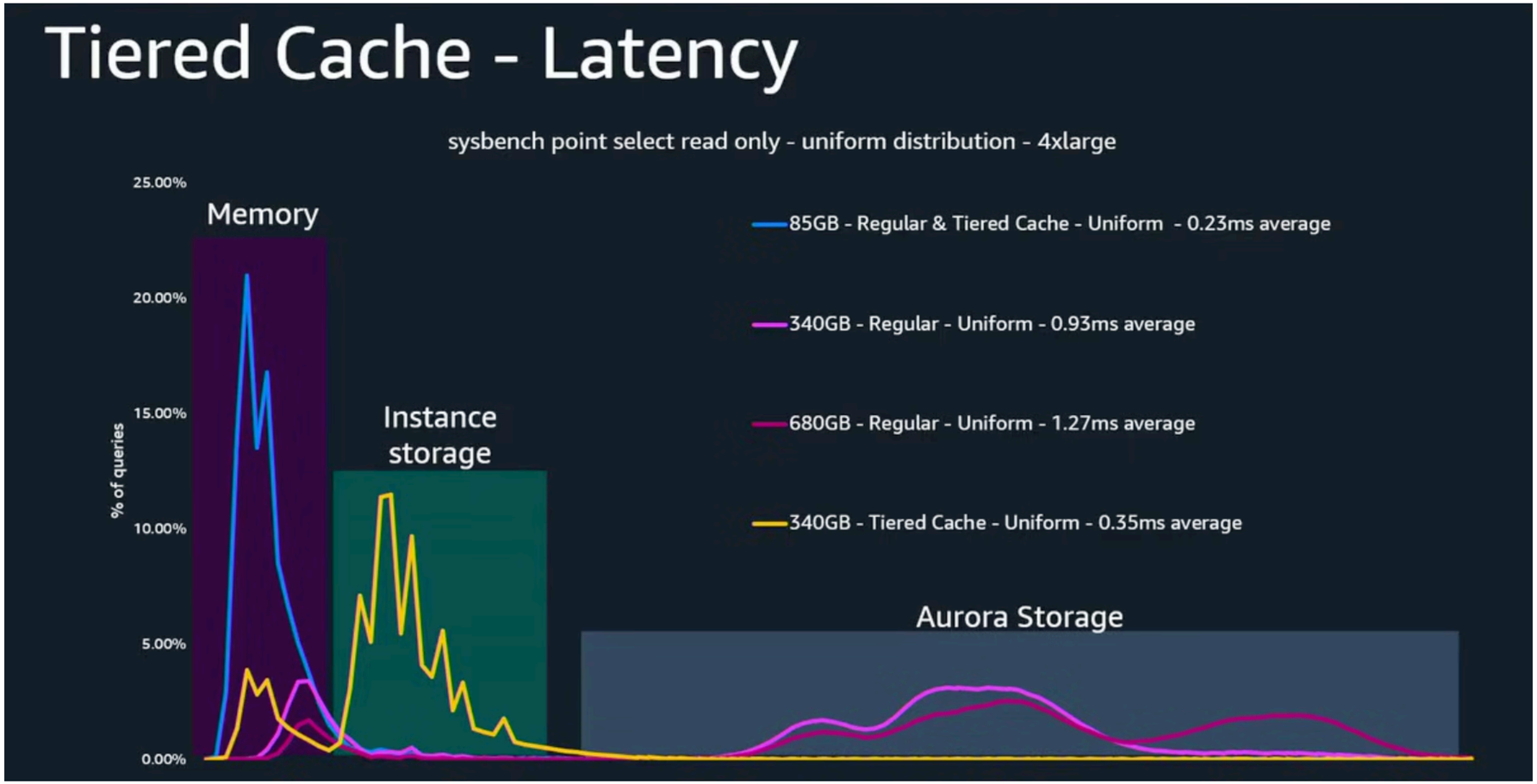
Blog Topics

Amazon Aurora
Amazon DocumentDB
Amazon DynamoDB
Amazon ElastiCache
Amazon Keyspaces (for Apache Cassandra)
Amazon Managed Blockchain
Amazon MemoryDB for Redis
Amazon Neptune
Amazon Quantum Ledger Database (Amazon QLDB)
Amazon RDS
Amazon Timestream
AWS Database Migration Service
AWS Schema Conversion Tool



AWS re:Invent

pganalyze



AWS re:Invent

pganalyze

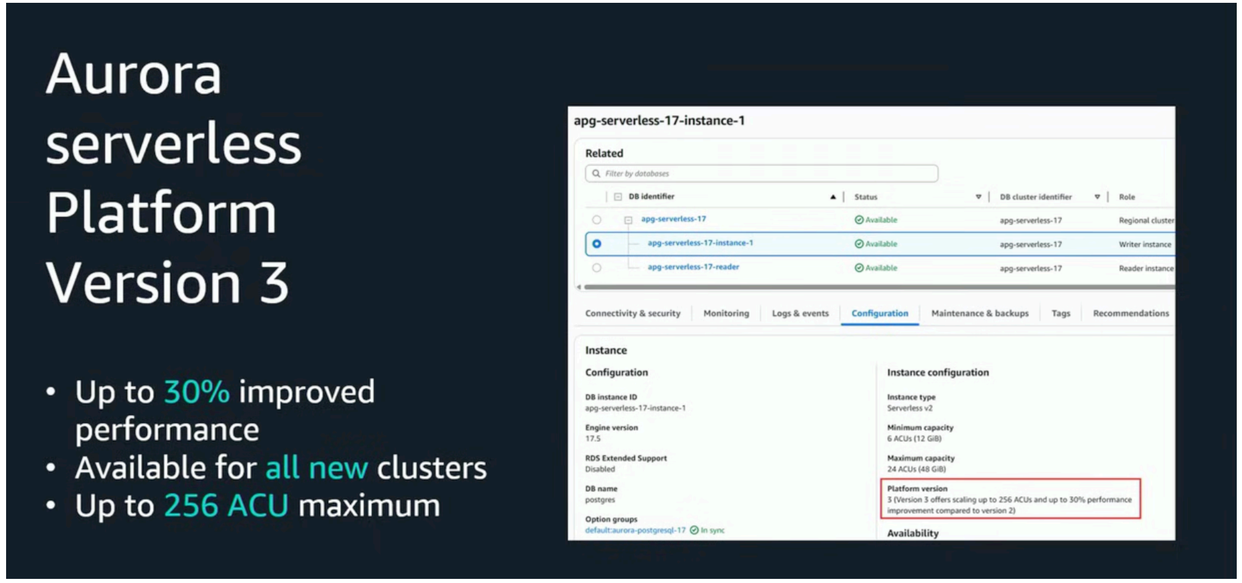
Monitoring DB instances that use Aurora Optimized Reads

You can monitor your queries that use Optimized Reads-enabled tiered cache with the EXPLAIN command as shown in the following example:

```
Postgres=> EXPLAIN (ANALYZE, BUFFERS) SELECT c FROM sbtest15 WHERE id=100000000

QUERY PLAN
------------------------------------------------------------------------------------------
 Index Scan using sbtest15_pkey on sbtest15  (cost=0.57..8.59 rows=1 width=121) (actual time=0.287..0.288 rows=
   Index Cond: (id = 100000000)
   Buffers: shared hit=3 read=2 aurora_orcache_hit=2
   I/O Timings: shared/local read=0.264
 Planning:
   Buffers: shared hit=33 read=6 aurora_orcache_hit=6
   I/O Timings: shared/local read=0.607
 Planning Time: 0.929 ms
 Execution Time: 0.303 ms
(9 rows)
Time: 2.028 ms
```

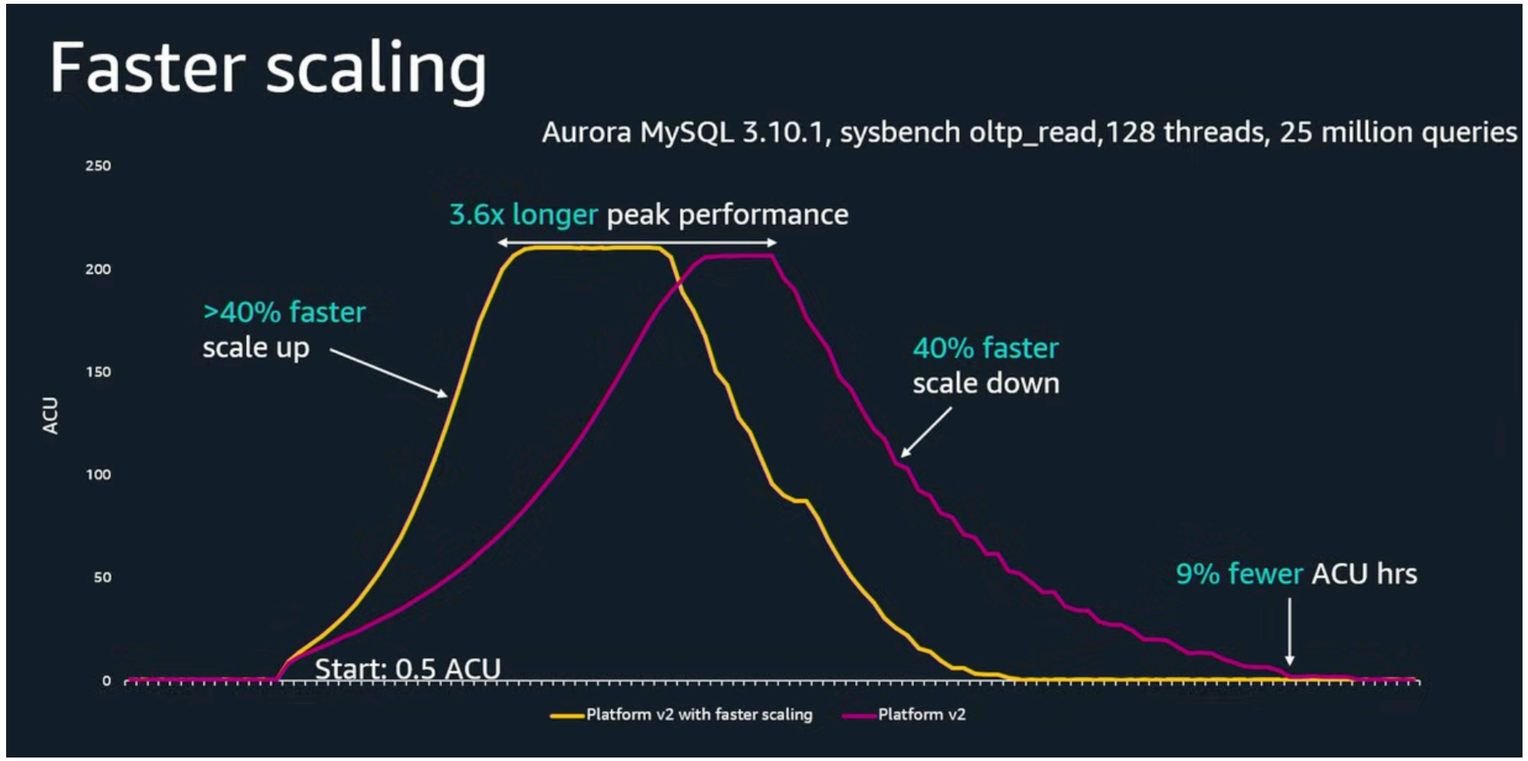
Aurora Serverless v2 Platform Version 3



Amazon Aurora Serverless v2 now offers up to 30% performance improvement

Posted on: Aug 7, 2025

Amazon Aurora Serverless v2 now offers up to 30% improved performance for databases running on the latest serverless platform version (version 3). Aurora Serverless v2 measures capacity in ACUs where each ACU is a combination of approximately 2 gibibytes (GiB) of memory, corresponding CPU, and networking. You specify the capacity range and the database scales within this range to support your application's needs. The version 3 serverless platform version supports scaling from 0 up to 256 Aurora Capacity Units (ACUs).



AWS re:Invent

pganalyze

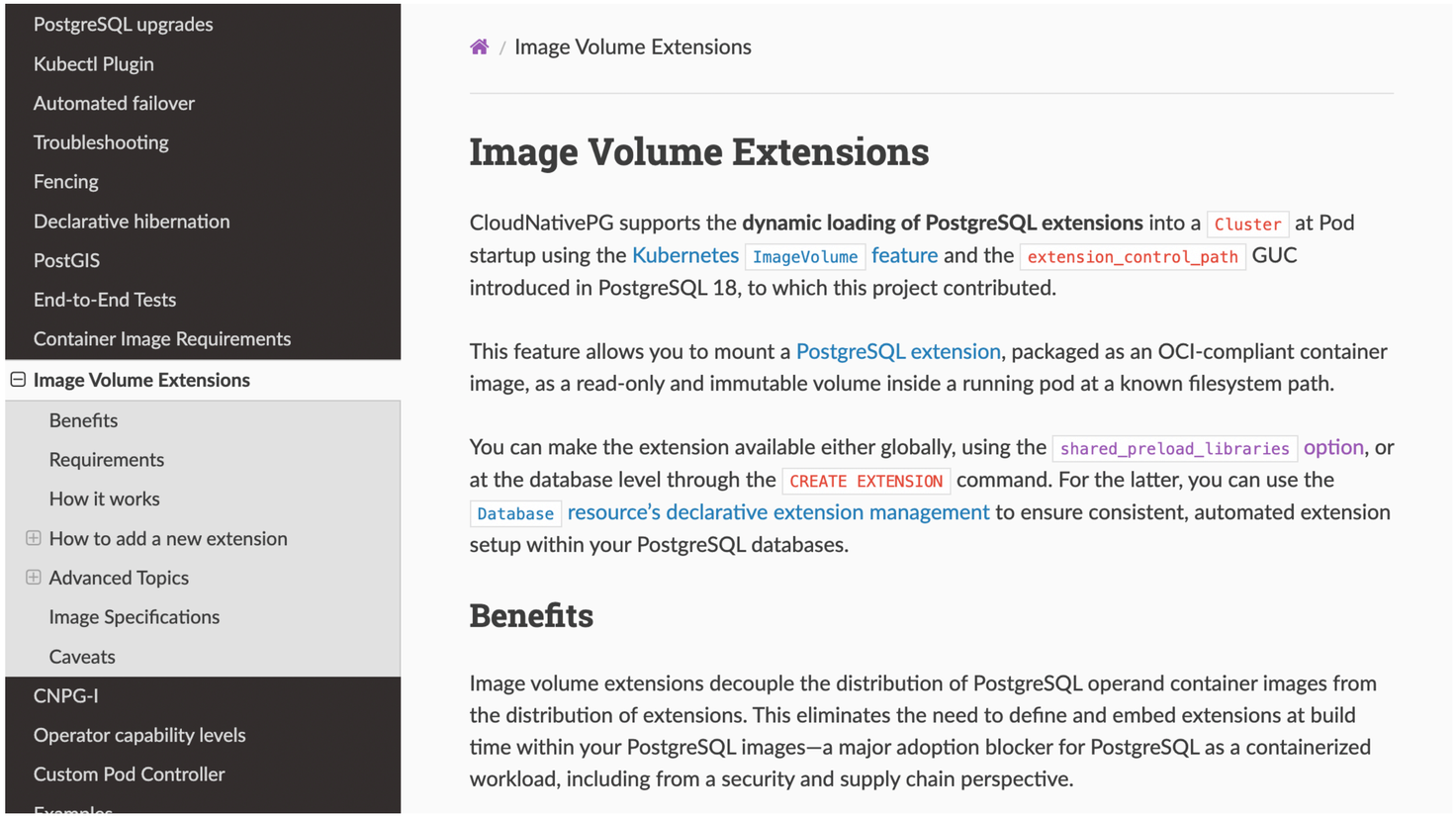
Extensibility

Extension availability is good, but not endless

Not available on RDS and Aurora:

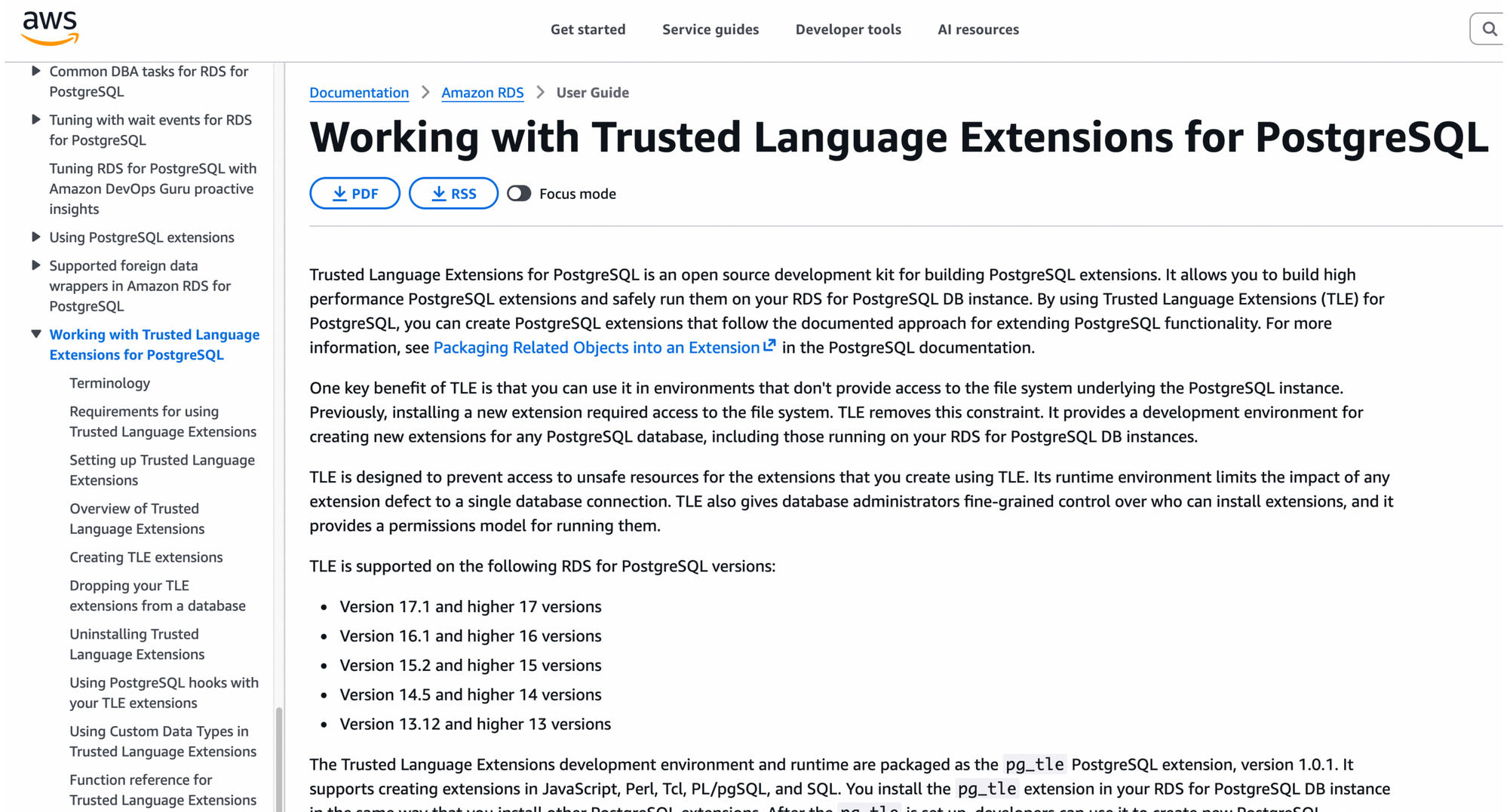
- pg_search (ParadeDB)
- Citus
- TimescaleDB

CNPG allows custom extension loading - to a running database server!



pganalyze

RDS and Aurora have pg_tle, but its very limited in practice



Example: There is still only a single hook available (passcheck)

Hooks reference for Trusted Language Extensions for PostgreSQL

Trusted Language Extensions for PostgreSQL supports PostgreSQL hooks. A *hook* is an internal callback mechanism available to developers for extending PostgreSQL's core functionality. By using hooks, developers can implement their own functions or procedures for use during various database operations, thereby modifying PostgreSQL's behavior in some way. For example, you can use a `passcheck` hook to customize how PostgreSQL handles the passwords supplied when creating or changing passwords for users (roles).

View the following documentation to learn about the passcheck hook available for your TLE extensions. To learn more about the available hooks including the client authentication hook, see Trusted Language Extensions hooks.

Password-check hook (passcheck)

The `passcheck` hook is used to customize PostgreSQL behavior during the password-checking process for the following SQL commands and `psql` metacommand.

- `CREATE ROLE username ...PASSWORD` – For more information, see CREATE ROLE in the PostgreSQL documentation.
- `ALTER ROLE username...PASSWORD` – For more information, see ALTER ROLE in the PostgreSQL documentation.
- `\password username` – This interactive `psql` metacommand securely changes the password for the specified user by hashing the password before transparently using the `ALTER ROLE ... PASSWORD` syntax. The metacommand is a secure wrapper for the `ALTER ROLE ... PASSWORD` command, thus the hook applies to the behavior of the `psql` metacommand.

For an example, see Password-check hook code listing.

Contents

Related resources

Amazon RDS API Reference

AWS CLI commands for Amazon RDS

SDKs & Tools

Recommended tasks

How to

Use Oracle SQL*Loader and PostgreSQL pg_dump to load data

Set up Trusted Language Extensions for PostgreSQL

Use cdk watch to continuously deploy and hotswap CDK changes

Configure automated backups for Amazon RDS databases

Create Resource Explorer resources with CloudFormation

Recently added to this guide

Amazon RDS supports **PL/Rust, Aurora does not**

Using PL/Rust to write PostgreSQL functions in the Rust language

PL/Rust is a trusted Rust language extension for PostgreSQL. You can use it for stored procedures, functions, and other procedural code that's callable from SQL. The PL/Rust language extension is available in the following versions:

- RDS for PostgreSQL 17.1 and higher 17 versions
- RDS for PostgreSQL 16.1 and higher 16 versions
- RDS for PostgreSQL 15.2-R2 and higher 15 versions
- RDS for PostgreSQL 14.9 and higher 14 versions
- RDS for PostgreSQL 13.12 and higher 13 versions

For more information, see PL/Rust on GitHub.

Topics

Setting up PL/Rust

To install the plrust extension on your DB instance, add plrust to the `shared_preload_libraries` parameter in the DB parameter group associated with your DB instance. With the plrust extension installed, you can create functions.

To modify the `shared_preload_libraries` parameter, your DB instance must be associated with a custom parameter group. For information

Related resources

- Amazon RDS API Reference
- AWS CLI commands for Amazon RDS
- SDKs & Tools

Recommended tasks

How to

- Replicate AWS Secrets Manager secrets across Regions
- Upgrade and use the PLV8 extension for PostgreSQL
- Migrate PostgreSQL database from EC2 to RDS using pglogical
- Migrate Oracle functions to PostgreSQL using extensions
- Transition objects between Amazon S3 storage classes

Debuggability & Community Support

"Aurora is not Postgres"

Aurora is heavily modified Postgres.

The community does not know Aurora,
and often won't (or can't) help

@phil - are those 48-72 hrs are for large databases, just trying to relate. And how much app is down for during that time.

phil Oct 16th at 6:03 AM

I'll be honest, some of them don't seem very large to me (< 1TB). I've tried asking AWS for more details on why the process goes the way it does, but they haven't really shared

I will say that most of the upgrades we see are less than 10 hours, but it really isn't uncommon to have them go several days, and each time we've reached out to AWS support and been told we just have to wait and let it run

And yes, the cluster is completely unavailable during the upgrade. So in those cases we are either doing straight logical replication to a new cluster we provision at the higher version, or a Blue/Green style - where we set up logical replication first, upgrade one side, and wait for the replication lag to clear up

Ants Aasma Dec 2nd at 11:03 AM

I have little experience with Aurora, but checking out the graphs, commit latency jumping to 100ms and staying steady at that level suggests some kind of transactions per second throughput limit in Aurora. Feels like Aurora has no group commit functionality or maybe a separate limit for WAL throughput.

Alex Theodore Dec 2nd at 10:11 AM

all of these terms and graphs are really specific to aurora... I honestly don't exactly know how to interpret them

Aurora does have novel functionality that is not upstreamed

Improving query performance using adaptive join

Overview

Adaptive join is a preview feature in Aurora PostgreSQL 17.4 that helps improve query performance. This feature is disabled by default, but you can enable it using Global User Configuration (GUC) parameters. Since this is a preview feature, the default parameter values might change. When enabled, adaptive join helps optimize query performance by dynamically switching from a nested loop join to a hash join at runtime. This switch occurs when the PostgreSQL optimizer has incorrectly chosen a nested loop join due to inaccurate cardinality estimates.

Configuring adaptive join

You can control adaptive join using these three GUC parameters:

Adaptive join configuration parameters

GUC parameter	Description	Default and configuration options
apg_adaptive_join_crossover_multiplier	This multiplier works with the *row crossover point* to determine when to switch from a nested loop to a hash join. The row crossover point	Controls whether Adaptive Join is enabled • Default value: -1 (disabled) • Valid range: -1 to DBL_MAX

Both RDS and Aurora
don't allow system access,
but sometimes you do
need a "perf profile"

Both RDS and Aurora
don't allow system access,
but sometimes you do
need a "perf profile"

Profiling with perf

perf is a utility set added to Linux kernel 2.6.31. A quick example showing what perf outpu

Contents [hide]

How to profile

perf offers two major modes: record then report, or real-time "top" mode. Both are useful in

TIP: **On many systems where -g is shown below, you may have to write --call-graph**

Record then report

Recording data

To profile the system for a specific length of time, for example 60 seconds:

```
perf record -a -g -s sleep 60
```

Planning performance problem (67626.278ms)

For the slow planning case that I saw, the slow process was almost entirely in this call stack (captured with perf record --call-graph):

...
index_fetch_heap
index_getnext
get_actual_variable_range
ineq_histogram_selectivity
scalarineqsel
mergejoinscansel
initial_cost_mergejoin
try_mergejoin_path
add_paths_to_joinrel
make_join_rel
join_search_one_level
standard_join_search
make_one_rel
query_planner
...

Planning time is time-consuming

Unsurprisingly, I also see planning as slower than execution, but with a ratio of about planning being 12x slower than execution vs the reported ~18x.

Planning Time: 0.581 ms
Execution Time: 0.048 ms

Nothing alarming in perf top of executing the query in pgbench with -M simple. I think this confirms the problem is just with expectations.

5.09%	postgres	[.] AllocSetAlloc
2.99%	postgres	[.] SearchCatCacheInternal
2.52%	postgres	[.] palloc
2.38%	postgres	[.] expression_tree_walker_impl
1.82%	postgres	[.] add_path_precheck
1.78%	postgres	[.] add_path
1.73%	postgres	[.] MemoryContextAllocZeroAligned
1.63%	postgres	[.] base_yyparse
1.61%	postgres	[.] CatalogCacheComputeHashValue
1.38%	postgres	[.] try_nestloop_path
1.36%	postgres	[.] stack_is_too_deep
1.33%	postgres	[.] add_paths_to_joinrel
1.19%	postgres	[.] core_yylex
1.18%	postgres	[.] lappend
1.15%	postgres	[.] initial_cost_nestloop
1.13%	postgres	[.] hash_search_with_hash_value
1.01%	postgres	[.] palloc0
0.95%	postgres	[.] get_memoize_path
0.90%	postgres	[.] equal
0.88%	postgres	[.] get_eclass_for_sort_expr
0.81%	postgres	[.] compare_pathkeys
0.80%	postgres	[.] bms_is_subset
0.77%	postgres	[.] ResourceArrayRemove

Monitoring

Core Postgres stats views and query monitoring extensions are supported on RDS and Aurora
(pg_stat_statements, auto_explain, etc)

Plan Statistics

aurora_plan_stats
captures plans over time

```
SELECT planid, calls, mean_exec_time FROM aurora_stat_plans(true) WHERE queryid = -99291
```

```
   planid    |  calls  |   mean_exec_time
-------------+---------+--------------------
 -1742045606 | 2556555 |   144.906537411382
  1209720180 |   90376 | 18.398387902275076
  1152279781 |     546 | 3.0803269890109903
(3 rows)
```

```
SELECT planid, plan_type, plan_captured_time, explain_plan FROM aurora_stat_plans(true)
```

```
-[ RECORD 1 ]------+-------------------------------------------------------------------------
planid             | -1742045606
plan_type          | estimate
plan_captured_time | 2024-11-14 08:35:28.182251+00
explain_plan       | Hash Join  (cost=60.00..100.24 rows=523 width=35)
                   |   Hash Cond: (o.product_id = p.product_id)
                   |   ->  Hash Join  (cost=30.50..69.36 rows=523 width=28)
                   |         Hash Cond: (o.customer_id = c.customer_id)
                   |         ->  Seq Scan on orders o  (cost=0.00..37.48 rows=523 width=2
                   |               Filter: (order_date > (now() - '14 days'::interval))
                   |         ->  Hash  (cost=18.00..18.00 rows=1000 width=16)
                   |               ->  Seq Scan on customers c  (cost=0.00..18.00 rows=1000
                   |   ->  Hash  (cost=17.00..17.00 rows=1000 width=15)
                   |         ->  Seq Scan on products p  (cost=0.00..17.00 rows=1000 widt
```

pganalyze

Introducing Postgres Plan Statistics in pganalyze for Amazon Aurora

By **Keiko Oda**
November 21, 2024

In this article:

At pganalyze we've offered query performance monitoring of Postgres databases for many years now, helping companies at scale ensure their Postgres database is performant and queries are as fast as possible. One common story we hear when it comes to analyzing Postgres performance, and identifying the root cause of slowdowns is: Has my query plan changed?

Recently Amazon Aurora, the highly scalable AWS PostgreSQL service, has made execution plan data more readily available by introducing `aurora_stat_plans`, a function now integrated with pganalyze to automatically collect plan statistics for all queries.

While pganalyze already offers robust query performance analysis—allowing you to identify slow queries, see how much system resources they're consuming, and dive deeper into why they're running inefficiently—this new integration gives our users on Amazon Aurora access to execution plans not usually retained by Postgres, without measurable overhead.

Today we're excited to introduce the new Plan Statistics feature in pganalyze, initially available on Amazon Aurora, with plans to expand it for all database servers going forward. This function is turned on by default in Amazon Aurora versions 14.10, 15.5, and higher, providing immediate access to enhanced insights without any additional configuration.

The challenge collecting Postgres Plan Statistics

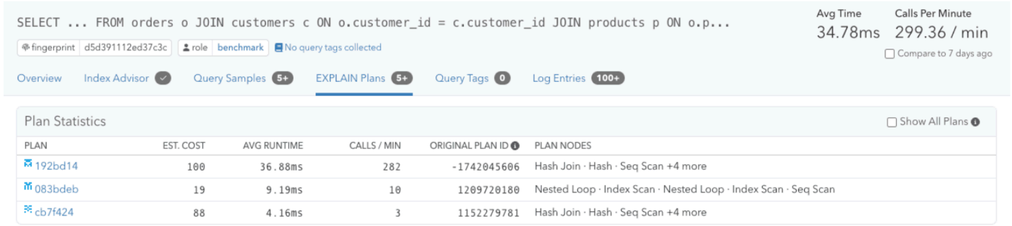
While PostgreSQL's statistics collector offers information about database activity, it lacks detailed execution plans for queries. With `pg_stat_statements`, PostgreSQL tracks statistics on a per-query basis—but it doesn't retain the execution plans that explain why certain queries perform poorly, or split out statistics based on different plans chosen for the same query. Tools like `EXPLAIN` and `EXPLAIN ANALYZE` can help investigate individual queries, and `auto_explain` can help track plans for outlier executions, but it isn't practical to run continuously on every single query because it would introduce too much overhead.

Back when `pg_stat_statements` was first developed, an alternate extension, pg_stat_plans was created as well, that allowed query plan tracking in addition to query statistics. The idea is simple: Instead of tracking a `queryid`, track the `planid`, that differentiates different query plans for the same query.

pganalyze

Comprehensive Execution Plan Collection

pganalyze automatically uses the `aurora_stat_plans` function to collect execution plans for all queries without manual intervention or performance impact.



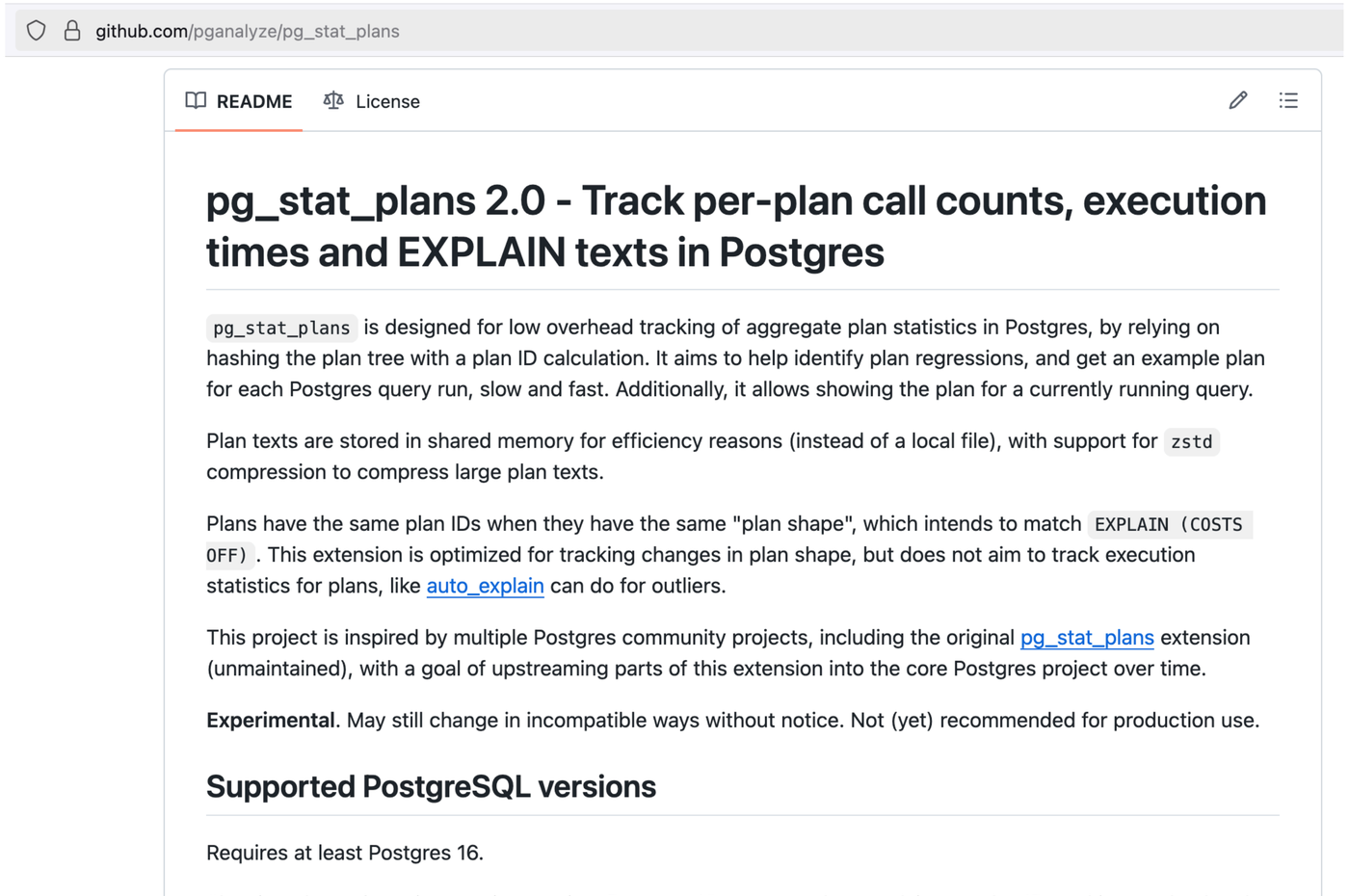
Note: pganalyze includes the original plan ID in addition to the Plan Fingerprint calculated by pganalyze. The original plan ID can be used with Aurora Query Plan Management, e.g. by calling the set_plan_status function, to mark a particular plan as preferred.

Historical Execution Plan Analysis

Execution plans are stored over time, allowing you to compare past and present plans for the same queries. This helps in identifying when and why a query's performance may have changed, or whether certain query plans perform worse than others for the same query:

pganalyze

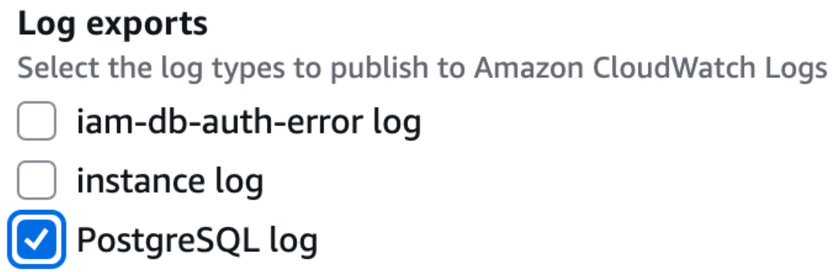
pg_stat_plans
does the same, for any Postgres (except RDS, yet)



pganalyze

Different ways to get logs
(you don't need to pay for CloudWatch exports!)

CloudWatch Log Export can be expensive



For a database we maintain, this costs $490 per month (for a single RDS instance)

pganalyze

RDS APIs are free!

Reading log file contents using REST

Amazon RDS provides a REST endpoint that allows access to DB instance log files. This is useful if you need to write an application to stream Amazon RDS log file contents.

The syntax is:

```
GET /v13/downloadCompleteLogFile/DBInstanceIdentifier/LogFileName HTTP/1.1
Content-type: application/json
host: rds.region.amazonaws.com
```

The following parameters are required:

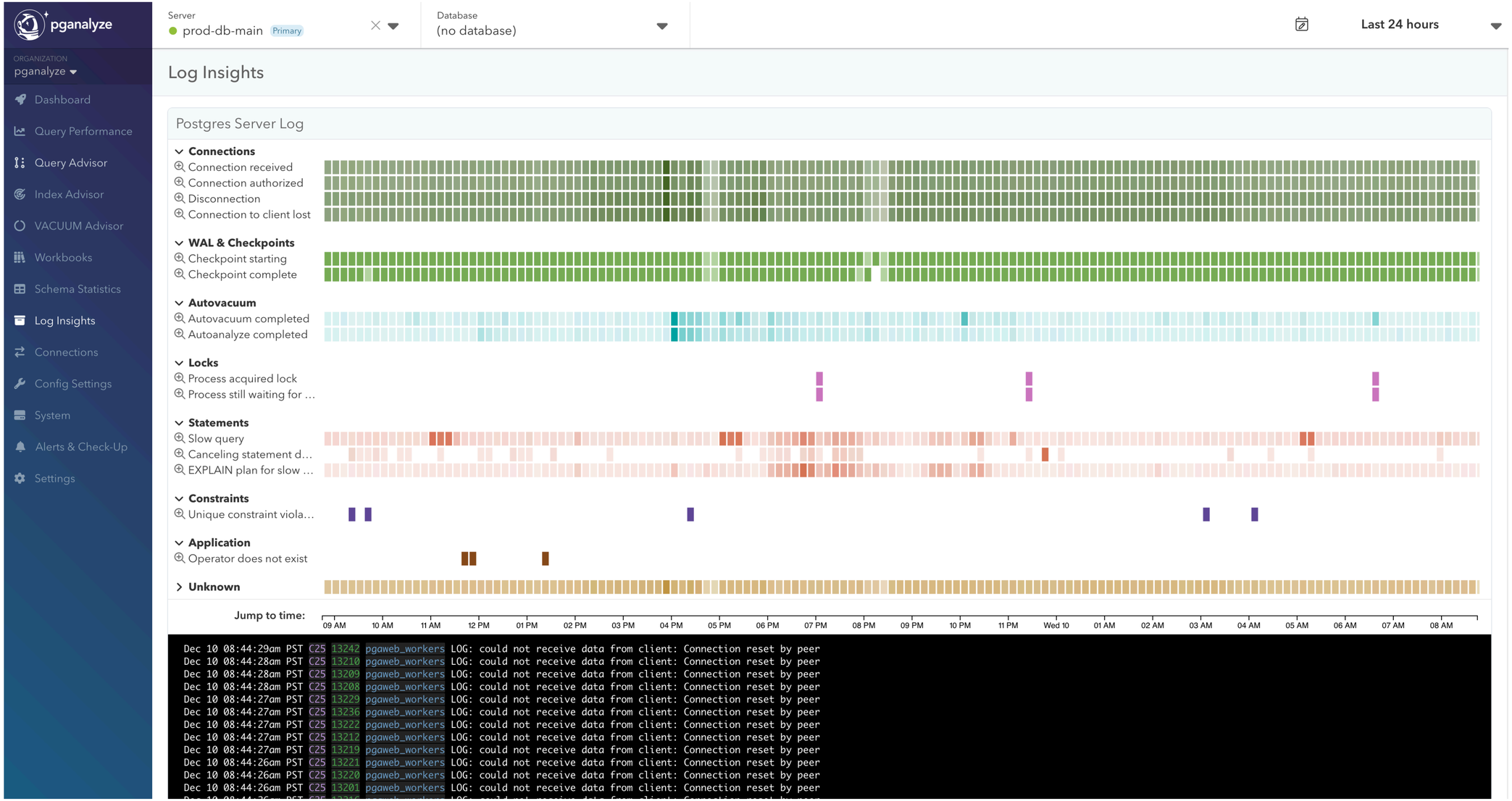
- *DBInstanceIdentifier* —the name of the DB instance that contains the log file you want to download.
- *LogFileName* —the name of the log file to be downloaded.

The response contains the contents of the requested log file, as a stream.

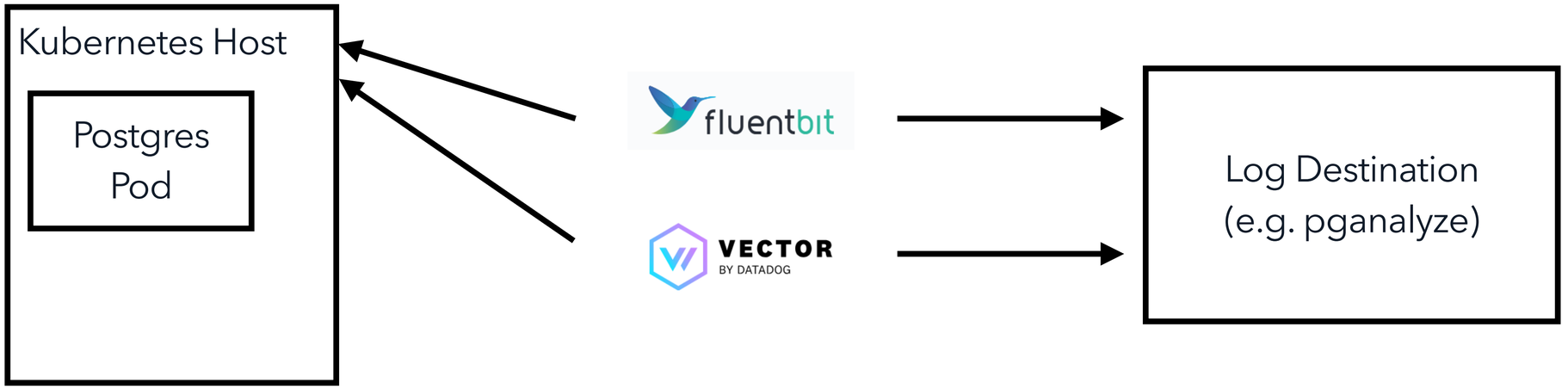
The following example downloads the log file named *log/ERROR.6* for the DB instance named *sample-sql* in the *us-west-2* region.

```
GET /v13/downloadCompleteLogFile/sample-sql/log/ERROR.6 HTTP/1.1
host: rds.us-west-2.amazonaws.com
X-Amz-Security-Token: AQoDYXdzEIH//////////wEa0AIXLhngC5zp9CyB1R6abwKrXHVR5efnAVN3XvR7IwqKYalFSn6UyJuEFTft9nObg
X-Amz-Date: 20140903T233749Z
X-Amz-Algorithm: AWS4-HMAC-SHA256
X-Amz-Credential: AKIADQKE4SARGYLE/20140903/us-west-2/rds/aws4_request
X-Amz-SignedHeaders: host
X-Amz-Content-SHA256: e3b0c44298fc1c229afbf4c8996fb92427ae41e4649b934de495991b7852b855
X-Amz-Expires: 86400
```

pganalyze Log Insights uses RDS APIs directly

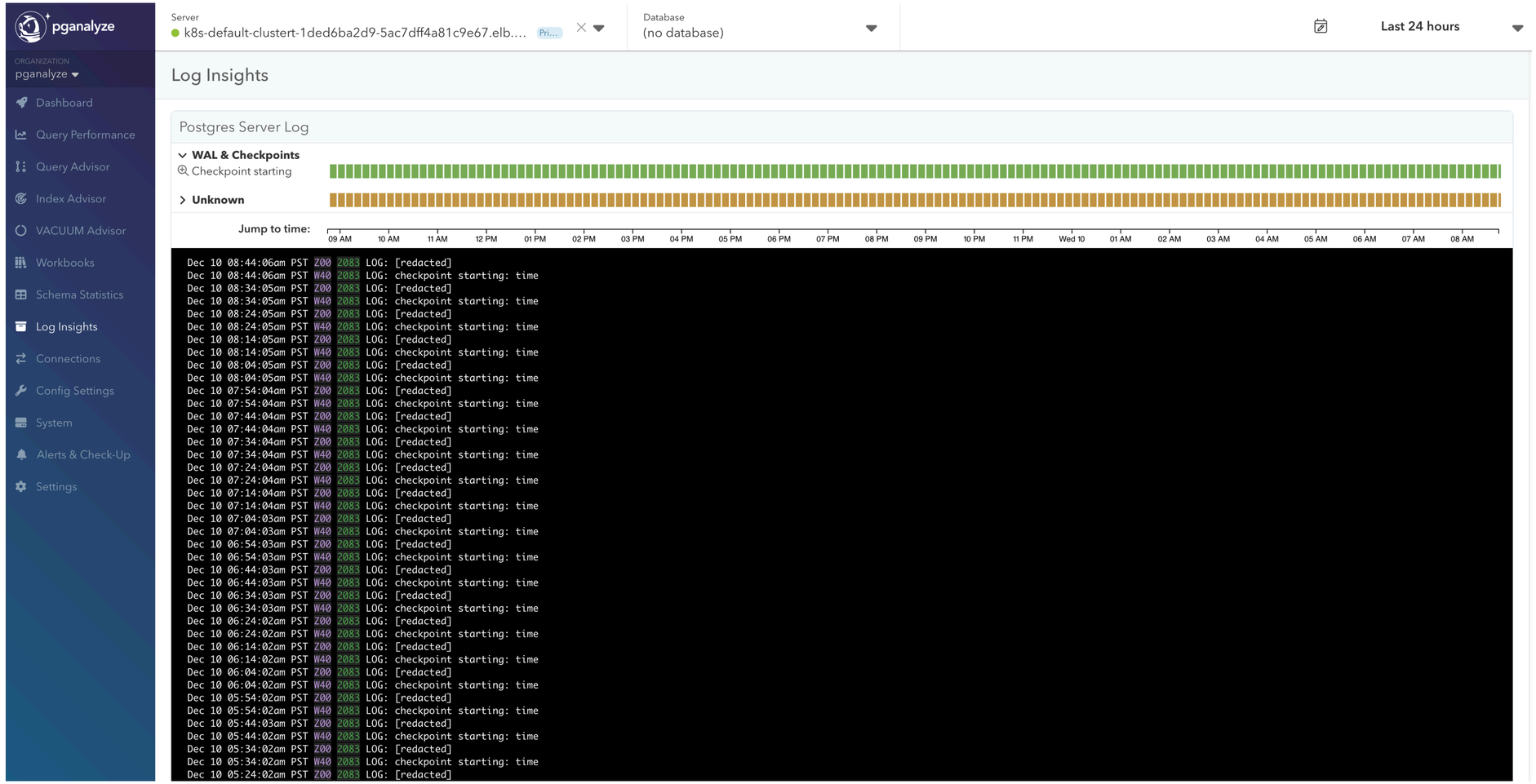


CNPG logs can be extracted via standard K8S mechanisms



pganalyze

pganalyze Log Insights works with CNPG too!



pganalyze

Database Insights
is the new Performance Insights

AWS Performance Insights has been deprecated: What to know about CloudWatch Database Insights

By **Lukas Fittl**
November 06, 2025

In this article:

At the end of November 2025, AWS is deprecating the built-in performance monitoring tool in Amazon RDS and Aurora, Performance Insights. AWS customers are asked to migrate to the new Database Insights functionality in CloudWatch by June 2026, with AWS noting:

> We recommend upgrading any database instance that will be impacted by this change before June 30, 2026. If no action is taken, your database instances will default to the standard mode of Database Insights after that date. As a result, you may lose access to your performance data history beyond seven days and to the execution plan and on-demand analysis features.

Since we have many customers using built-in AWS tools together with pganalyze, we've heard many questions on what this migration is about, and how the functionality between the two AWS tools differs. In this blog post we're going to explore key functionality differences, the difference between Standard and Advanced Mode in Database Insights, and hidden cost drivers in CloudWatch Logs that are important to know about when migrating.

Whilst pganalyze can supplement or replace AWS CloudWatch Database Insights, in this post we're fully focused on comparing the tools offered by AWS. If you want to know how pganalyze compares for your organization's use cases, reach out to us for a demo.

The essence of Performance Insights: Active Session History

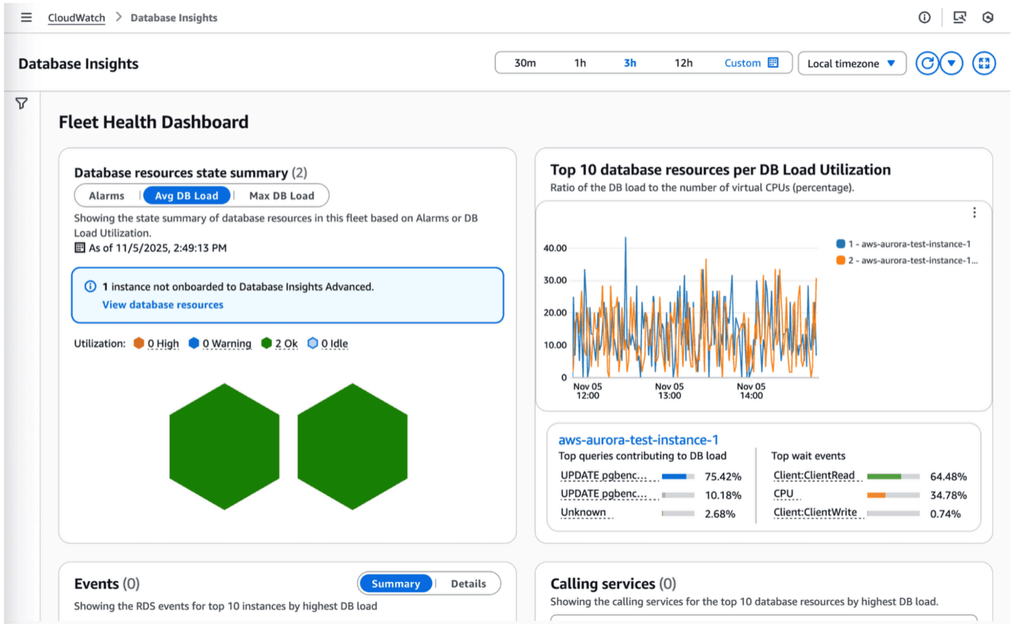
One of the key benefits of AWS Performance Insights was its ability to capture wait events and active query information using sampling. For Postgres specifically, this mainly utilizes information from `pg_stat_activity`, which we can also query directly like this:

```
SELECT COUNT(*), state, wait_event_type, wait_event FROM pg_stat_activity WHERE state <>
```

pganalyze

New: Fleet-wide overview

One of the new functionalities introduced by Database Insights is the all-up view of database instances across an AWS account. This is only supported for instances configured with the paid tier (Advanced Mode), and has two key functionalities. It can surface instances that have an active CloudWatch alarm firing, and highlight them in red in a hexagon-based visualization:

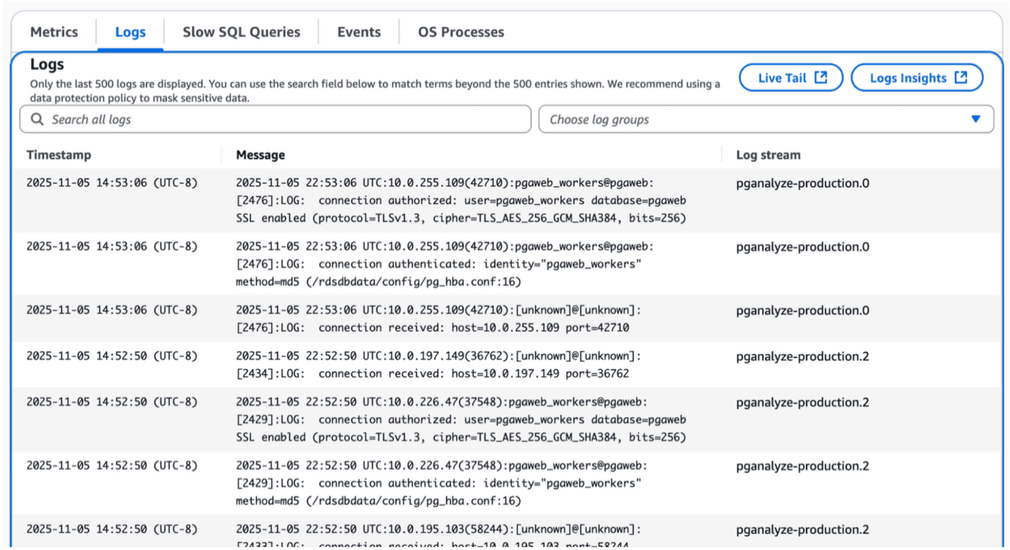


Additionally, the fleet view also surfaces top activity across all instances, like instances consuming most CPU, and top queries across all instances.

Detailed analysis still requires drilling down to individual instances, but the new fleet view makes it easier to surface the Top 10 instances or active alerts firing.

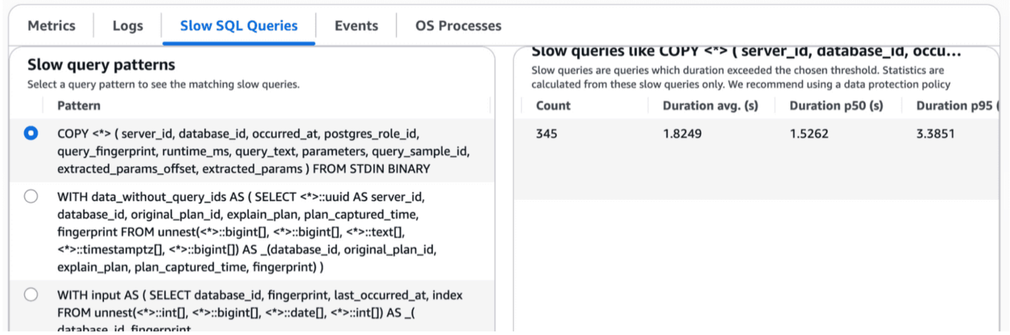
New: Log Viewer and Slow Query Log

With CloudWatch Log exports enabled, Database Insights can also surface data previously available in the Logs view:



This functionality is similar to the built-in log viewer in other platforms like Google CloudSQL, but requires using the separately charged CloudWatch log export functionality. The existing log view/downloads in the Aurora/RDS console remain available and free of charge.

Additionally, this view can pull out only slow query log events:



	Performance Insights	Database Insights (Standard)	Database Insights (Advanced)
Wait Event Analysis for Last 7 days	Yes	Yes	Yes
Retention Time	Customizable (up to 24 mo.)	7 days (not customizable)	15 months
Operating System Metrics (RDS Enhanced Monitoring)	Yes	No	Yes
Fleet Management View	No	No	Yes
Lock Analysis	No	No	Yes (Aurora Postgres only)
Query Plan Statistics	Yes (Aurora only)	No	Yes (Aurora only)
Per-Query Statistics	No	No	Yes
Slow Query Log	No	No	Yes (requires CloudWatch Log Export)
Performance Metrics Export to CloudWatch	Yes	No	Yes
Analysis of specific time periods	No	No	Yes

Pricing differences

The most drastic change in CloudWatch Database Insights are the charges for using Advanced Mode. The fee for enabling Advanced Mode are based on the instance size (number of vCPUs), and scales up with more vCPUs, quickly going beyond $100/month/instance on large production databases, as shown here for us-east-1:

Instance Size (vCPUs)	Monthly Charge for Database Insights (per instance)
2 vCPU	$18.25 per month
4 vCPU	$36.50 per month
8 vCPU	$73.00 per month
16 vCPU	$146.00 per month
32 vCPU	$292.00 per month
64 vCPU	$584.00 per month

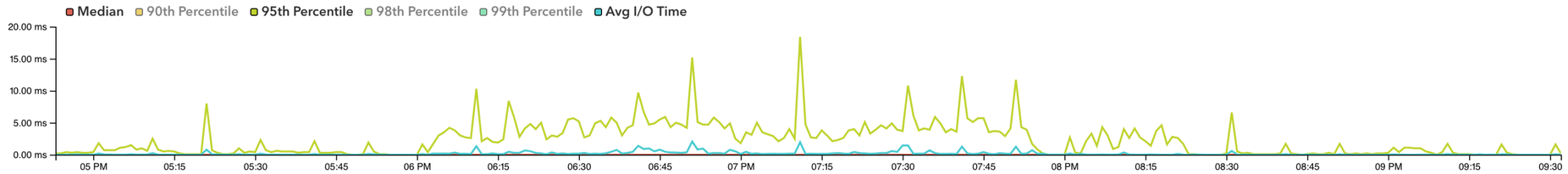
pganalyze vs Database Insights

Compare pganalyze to CloudWatch Database Insights	Database Insights	pganalyze
Top-Level Query Metrics	✓ For Top 25 Queries	✓ For All Queries (Unlimited)
Drill Into Per-Query Metrics	Only with Advanced Mode	✓
EXPLAIN Plan Statistics	Only with Advanced Mode and Amazon Aurora	✓
EXPLAIN Plan Samples with execution details	✗	✓
Automatic EXPLAIN ANALYZE through auto_explain	✗	✓
EXPLAIN Insights ("Why is this query slow?")	✗	✓
EXPLAIN Plan Comparison	Plan text side-by-side view for Plan Statistics	✓ Purpose-built diff of plan structure and metrics
Query Rewrite Recommendations	✗	✓
Index Recommendations	✗	✓
VACUUM Recommendations	✗	✓

pganalyze

pganalyze has a better drill-down experience

Database Queries



SELECT | INSERT, UPDATE, DELETE | DDL & other | Compare to 7 days ago | Search...

QUERY	ROLE	AVG TIME (MS)	7 DAY TREND	CALLS / MIN	7 DAY TREND	% OF ALL I/O	% OF ALL RUNTIME
SELECT * FROM (SELECT ... FROM schema_table_stats_packed…	pgaweb_workers +1	3.73ms	+3.51ms	20377.46	+18866.87	8.30%	18.70%
WITH combined_table_stats AS (...), stats_interval AS (.…	pgaweb_workers +1	10.05ms	-33.84ms	5695.39	+5695.06	1.96%	14.08%
SELECT ... FROM unpack_schema_table_stats(database_ids :…	pgaweb_workers +1	452.90ms	+422.42ms	73.94	+71.82	7.46%	8.24%
WITH input AS (...), matching_queries AS MATERIALIZED (.…	pgaweb_workers	0.94ms	+0.71ms	31459.70	+62.31	20.08%	7.28%
SELECT ... FROM unpack_schema_index_stats(database_ids :…	pgaweb_workers +1	18.83ms	+16.70ms	1226.90	+1208.93	0.00%	5.69%
SELECT * FROM (SELECT ... FROM schema_index_stats_packed…	pgaweb_workers +1	8.17ms	+7.82ms	2710.60	+1221.78	0.05%	5.45%
WITH ranked AS (...) SELECT * FROM ranked WHERE rn <= $1	pgaweb_workers	1,742.21ms	(no change)	8.95	(no change)	17.45%	3.84%
SELECT ... FROM queries, unnest($1::bigint[], $2::bigint…	pgaweb_workers	272.76ms	-749.47ms	51.23	+51.07	10.88%	3.44%
SELECT * FROM query_stats WHERE ...	pgaweb_workers +1	2.00ms	+1.88ms	5213.70	+2551.83	1.71%	2.57%
WITH input AS (...), existing_queries AS (...), update_q…	pgaweb_workers	15.15ms	+10.58ms	592.16	+22.64	6.30%	2.21%
SELECT ... FROM unpack_schema_table_stats(database_ids :…	pgaweb_workers +1	1.76ms	-3.10ms	5071.24	+5070.57	0.00%	2.20%

pganalyze has a better drill-down experience

pganalyze

pganalyze has a better drill-down experience

```
SELECT * FROM (SELECT ... FROM schema_table_stats_packed_35d WHERE ...) _ WHERE ...
```

fingerprint 557af0f1aa47a132 | role pgaweb_workers and 1 more

Avg Time 3.87ms | Calls Per Minute 23,468.84 / min

Compare to 7 days ago

Overview | Index Advisor ? | Query Samples 5+ | EXPLAIN Plans 5+ | Query Tags 0 | Log Entries 100+

Plan Statistics

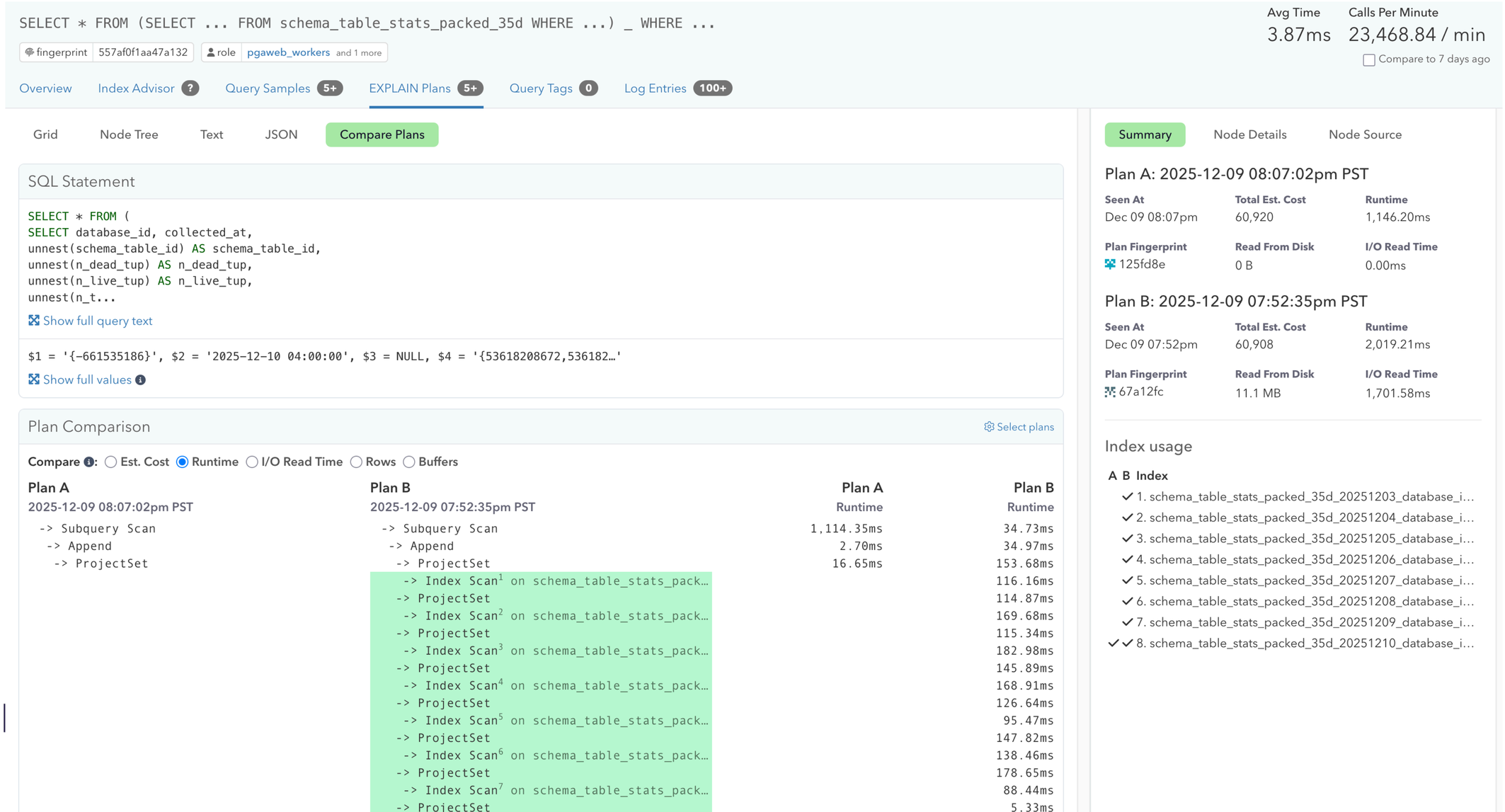
PLAN	EST. COST	AVG RUNTIME	PLAN SAMPLES	PLAN NODES
125fd8e	60,928	1,123.12ms	8	Subquery Scan · Append · ProjectSet · Index Scan
67a12fc	60,838	2,807.35ms	206	Subquery Scan · Append · ProjectSet +15 more
e511aa5	60,872	4,347.64ms	1	Subquery Scan · Append · ProjectSet +29 more

Plan Samples (215)

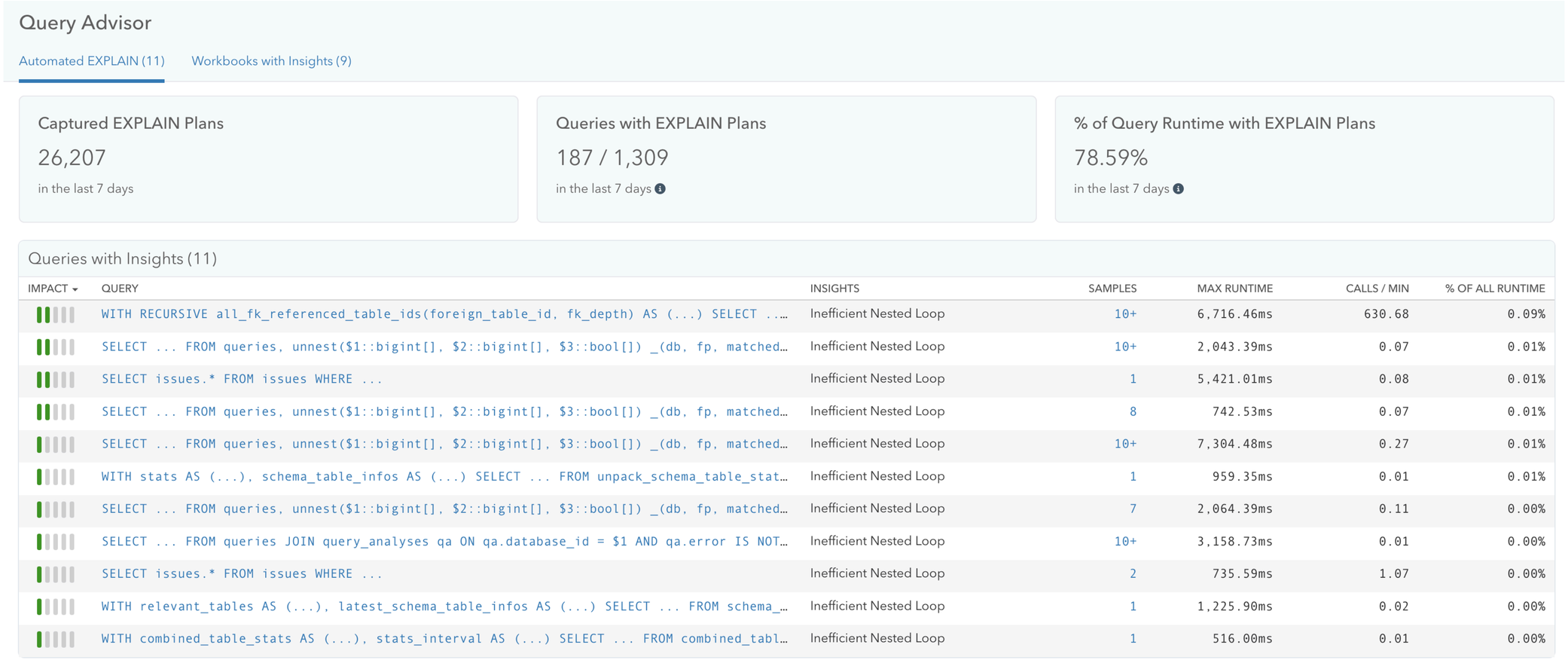
Search plan fingerprint...

EXECUTED AT	PLAN	EST. COST	RUNTIME	I/O READ TIME		READ FROM DISK	PLAN NODES
2025-12-09 08:32:01pm PST	125fd8e	60,957	1,051.79ms	0.00ms	0%	0 B	Subquery Scan · Append · ProjectSet · Index Scan
2025-12-09 08:26:27pm PST	125fd8e	60,945	1,017.52ms	0.00ms	0%	0 B	Subquery Scan · Append · ProjectSet · Index Scan
2025-12-09 08:19:07pm PST	125fd8e	60,932	1,053.37ms	0.00ms	0%	0 B	Subquery Scan · Append · ProjectSet · Index Scan
2025-12-09 08:15:58pm PST	125fd8e	60,932	1,128.98ms	0.00ms	0%	0 B	Subquery Scan · Append · ProjectSet · Index Scan
2025-12-09 08:14:46pm PST	125fd8e	60,932	1,046.97ms	0.00ms	0%	0 B	Subquery Scan · Append · ProjectSet · Index Scan
2025-12-09 08:09:32pm PST	125fd8e	60,920	1,014.36ms	0.00ms	0%	0 B	Subquery Scan · Append · ProjectSet · Index Scan
2025-12-09 08:07:02pm PST	125fd8e	60,920	1,146.20ms	0.00ms	0%	0 B	Subquery Scan · Append · ProjectSet · Index Scan
2025-12-09 07:52:35pm PST	67a12fc	60,908	2,019.21ms	1,701.58ms	84%	11.1 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:51:41pm PST	67a12fc	60,906	2,126.26ms	1,369.81ms	64%	32.8 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:51:14pm PST	67a12fc	60,905	1,521.47ms	748.09ms	49%	17.1 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:50:48pm PST	67a12fc	60,905	1,322.54ms	1,141.54ms	86%	11 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:50:23pm PST	67a12fc	60,901	2,588.29ms	2,419.39ms	93%	12.6 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:49:44pm PST	67a12fc	60,893	1,874.47ms	1,137.06ms	61%	28.8 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:49:20pm PST	67a12fc	60,893	1,489.07ms	950.98ms	64%	19.6 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:48:09pm PST	67a12fc	60,893	2,000.20ms	1,257.21ms	63%	27.6 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:47:48pm PST	67a12fc	60,893	2,794.73ms	2,063.54ms	74%	19.4 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:47:08pm PST	67a12fc	60,893	4,314.22ms	3,526.51ms	82%	43.6 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:46:58pm PST	67a12fc	60,893	1,824.43ms	1,116.17ms	61%	23.3 MB	Subquery Scan · Append · ProjectSet +15 more
2025-12-09 07:46:36pm PST	67a12fc	60,893	1,307.21ms	737.15ms	56%	16.3 MB	Subquery Scan · Append · ProjectSet +15 more

pganalyze has a better drill-down experience



pganalyze has Query Advisor



IMPACT	QUERY	INSIGHTS	SAMPLES	MAX RUNTIME	CALLS / MIN	% OF ALL RUNTIME
	WITH RECURSIVE all_fk_referenced_table_ids(foreign_table_id, fk_depth) AS (...) SELECT ..…	Inefficient Nested Loop	10+	6,716.46ms	630.68	0.09%
	SELECT ... FROM queries, unnest($1::bigint[], $2::bigint[], $3::bool[]) _(db, fp, matched…	Inefficient Nested Loop	10+	2,043.39ms	0.07	0.01%
	SELECT issues.* FROM issues WHERE ...	Inefficient Nested Loop	1	5,421.01ms	0.08	0.01%
	SELECT ... FROM queries, unnest($1::bigint[], $2::bigint[], $3::bool[]) _(db, fp, matched…	Inefficient Nested Loop	8	742.53ms	0.07	0.01%
	SELECT ... FROM queries, unnest($1::bigint[], $2::bigint[], $3::bool[]) _(db, fp, matched…	Inefficient Nested Loop	10+	7,304.48ms	0.27	0.01%
	WITH stats AS (...), schema_table_infos AS (...) SELECT ... FROM unpack_schema_table_stat…	Inefficient Nested Loop	1	959.35ms	0.01	0.01%
	SELECT ... FROM queries, unnest($1::bigint[], $2::bigint[], $3::bool[]) _(db, fp, matched…	Inefficient Nested Loop	7	2,064.39ms	0.11	0.00%
	SELECT ... FROM queries JOIN query_analyses qa ON qa.database_id = $1 AND qa.error IS NOT…	Inefficient Nested Loop	10+	3,158.73ms	0.01	0.00%
	SELECT issues.* FROM issues WHERE ...	Inefficient Nested Loop	2	735.59ms	1.07	0.00%
	WITH relevant_tables AS (...), latest_schema_table_infos AS (...) SELECT ... FROM schema_…	Inefficient Nested Loop	1	1,225.90ms	0.02	0.00%
	WITH combined_table_stats AS (...), stats_interval AS (...) SELECT ... FROM combined_tabl…	Inefficient Nested Loop	1	516.00ms	0.01	0.00%

pganalyze

pganalyze has Query Advisor

Insight 1 of 1: Inefficient Nested Loop

Found in plan: db5ccf2

Pattern Detected: Found a Nested Loop with a high row misestimate on the outer table. Try rewriting the query with a materialized CTE to use a different query plan.

Learn more about this insight.

Current Query

Impact

```
/*controller:graphql,action:graphql,line:/app/graphql/resolvers/get_issues.rb:70:in
`resolve',sentry_trace_id:4632bf07f60e42b1802a1284b4b5a2b5,traceparent:00-
ce9e6c37a708146fd1183598024d1e14-f356b8ba603da878-01,tracestate:pganalyze=t:1744904610.0694833*/

SELECT issues.*
FROM issues
WHERE
  issues.server_id = $1
  AND issues.server_id = ANY($2)
  AND (issues.database_id = ANY($3) OR issues.database_id IS NULL)
  AND id IN (
    SELECT issue_references.issue_id
    FROM issue_references
    WHERE
      issue_references.organization_id = $4
      AND issue_references.referent_type = $5
      AND issue_references.referent_id = $6
      AND issue_references.resolved_at IS NULL
  )
  AND issues.state = ANY($7)
```

Suggested Query Rewrite

Show Rewrite Steps

Create Workbook

```
/*controller:graphql,action:graphql,line:/app/graphql/resolvers/get_issues.rb:70:in
`resolve',sentry_trace_id:4632bf07f60e42b1802a1284b4b5a2b5,traceparent:00-
ce9e6c37a708146fd1183598024d1e14-f356b8ba603da878-01,tracestate:pganalyze=t:1744904610.0694833*/
WITH __issue_references AS MATERIALIZED (
  SELECT
    issue_references.id, issue_references.created_at,
    issue_references.resolved_at, issue_references.organization_id,
    issue_references.server_id, issue_references.issue_id,
    issue_references.referent_type, issue_references.referent_id,
    issue_references.details
  FROM issue_references
  WHERE
    issue_references.referent_type = $5
    AND issue_references.referent_id = $6
)
SELECT issues.*
FROM issues
WHERE
  issues.server_id = $1
  AND issues.server_id = ANY($2)
  AND (issues.database_id = ANY($3) OR issues.database_id IS NULL)
  AND id IN (
    SELECT issue_references.issue_id
    FROM __issue_references issue_references
    WHERE
      issue_references.organization_id = $4

      AND issue_references.resolved_at IS NULL
  )
  AND issues.state = ANY($7)
```

EXPLAIN Plan

Select plan

Show: Est. Cost Runtime Rows Buffers Reads Writes

pganalyze has Index Advisor

Index Advisor

Overview | Missing Indexes | Unused Indexes | Configure | Status

Total Data Size	Total Index Size	Table Writes	Avg. Index Write Overhead
5.2 TB	1.3 TB	801,239	0.31
▲ 114.2 GB		per minute	index bytes per table byte

Insights for Database (26)

INSIGHT	
8	Missing Indexes Based on query activity in last 7 days
18	Unused Indexes Not recently used (default 35 days)

Create Index (3)

Copy 3 commands

INDEX	SCAN COST CHANGE ▲	INDEX WRITE OVERHEAD	AFFECTED QUERIES
I9 btree (email)	-3,820	+0.05	2
I6 btree (auth_organization_id)	-3,619	+0.07	2
I10 btree (unconfirmed_email)	-1,910	+0.06	1

Keep Existing Indexes (5)

INDEX	INDEX WRITE OVERHEAD	AFFECTED QUERIES
I1 btree (id) primary key	0.02	35
I2 btree (auth_uid, auth_provider, COALESCE(auth_organization_id, '00000000-0000-00… unique	0.07	2
I3 btree (confirmation_token) unique	0.11	1
I4 btree (reset_password_token) unique	0.10	2
I5 btree (COALESCE(auth_organization_id, '00000000-0000-0000-0000-000000000000'::te… unique	0.04	0

pganalyze

pganalyze has VACUUM Advisor

VACUUM Advisor

Overview | Bloat | Freezing | Performance | Activity

Est. Table Bloat
168.3 GB 3%
based on 753 of 789 tables

Total Tables
789

Xmin Horizon Assigned
0.2
hours ago

Insufficient VACUUM Frequency

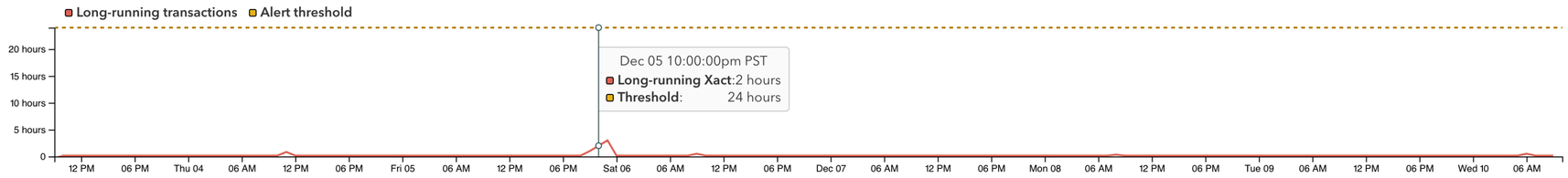
Detects when insufficient VACUUM frequency leads to new bloat in a table. It analyzes the table statistics from the past seven days and calculates the amount of table growth that could have been avoided if VACUUM had been performed more frequently. Detection works best after removing existing bloat on the table, e.g. through pg_repack. Learn more

Looks good (last checked 14 hours ago)

VACUUM Blocked by Xmin Horizon

Detects when the xmin horizon on a server was assigned too far in the past, preventing vacuum from performing necessary cleanup of dead rows. Learn more

Looks good (last checked 26 minutes ago)



Bloat-Related Config Settings

SETTING	CURRENT VALUE
autovacuum_vacuum_scale_factor	0.1
autovacuum_vacuum_threshold	50

pganalyze

Database Savings Plans

Introducing Database Savings Plans for AWS Databases

by Betty Zheng (郑予彬) | on 02 DEC 2025 | in Announcements, AWS re:Invent, Database, Launch, News | Permalink | Comments | Share

0:00 / 0:00

Voiced by Amazon Polly

Since Amazon Web Services (AWS) introduced Savings Plans, customers have been able to lower the cost of running sustained workloads while maintaining the flexibility to manage usage across accounts, resource types, and AWS Regions. Today, we're extending this flexible pricing model to AWS managed database services with the launch of Database Savings Plans, which help customers reduce database costs by up to 35% when they commit to a consistent amount of usage ($/hour) over a **1-year** term. Savings automatically apply each hour to eligible usage across supported database services, and any additional usage beyond the commitment is billed at on-demand rates.

As organizations build and manage data-driven and AI applications, they often use different database services, engines and deployment types, including instance-based and serverless options, to meet evolving business needs. Database Savings Plans provide the flexibility to choose how workloads run while maintaining cost efficiency. If customers are in the middle of a migration or modernization effort, they can switch database engines and adjust deployment types, such as from provisioned to serverless as part of ongoing cost optimization, while continuing to receive discounted rates. If a customer's business expands globally, they can also shift usage across AWS Regions and continue to benefit from the same commitment. By applying a consistent hourly commitment, customers can maintain predictable spend even as usage patterns evolve and analyze coverage and utilization using familiar cost management tools.

New Savings Plans

Each plan defines where pricing applies, the range of available discounts, and the level of flexibility provided across supported database engines, instance families, sizes, deployment options, or AWS Regions.

The hourly commitment automatically applies to all eligible usage regardless of Region, with support for Amazon Aurora, Amazon Relational Database Service (Amazon RDS), Amazon DynamoDB, Amazon ElastiCache, Amazon DocumentDB (with MongoDB compatibility), Amazon Neptune, Amazon Keyspaces (for Apache Cassandra), Amazon Timestream, and AWS Database Migration Service (AWS DMS). As new eligible database offerings, instance types, or Regions become available, Savings Plans will automatically apply to that usage.

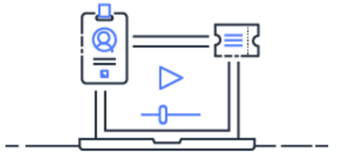
Discounts vary by deployment model and service type. Serverless deployments provide up to 35% savings compared to on-demand rates. Provisioned instances across supported database services deliver up to 20% savings. For Amazon DynamoDB and Amazon Keyspaces, on-demand throughput workloads receive up to 18% savings, and provisioned

Resources

Getting Started
What's New
Top Posts
Official AWS Podcast
Case Studies
AWS re:Post

Follow

Twitter
Facebook
LinkedIn
Twitch
RSS Feed
Email Updates



Reserved Instances (no upfront): 22-35% Savings

Database Savings Plans (no upfront): 20% Savings

Does not apply to storage charges, but
does apply to increased I/O Optimized instance costs

Database Savings Plans are a $-based commit,
not tied tied to region or instance class
(and they work with Serverless!)

Compute Savings Plans will offer a higher % savings
(~34%) for VM/K8S-based workloads

Thank you!

Try out pganalyze:

PGANALYZE.COM

Reach out for any questions:

lukas@pganalyze.com
