PostgreSQL Health check runbook

Revision 1.5, August 2024

Notices

Contents

Introduction

Runbook audience and overview

This runbook is designed for internal use by Solutions Architects, Specialist Sales, and other roles who work directly with customers to troubleshoot and solve performance concerns with Aurora PostgreSQL and RDS Postgres. It describes the recommended steps and best practices for evaluating the health of an existing PostgreSQL database, and finding areas of concern/improvement that both alleviate performance bottlenecks and ensure sustained appropriate performance while following existing best practices. This runbook can be used for:

1. Upon concern of PostgreSQL performance problems
2. Diagnosing an existing database schema in an effort to gather data for someone else to continue troubleshooting

The runbook is specifically written for the PostgreSQL engine, and will focus on core PostgreSQL concepts. It can be used for customers using RDS Postgres, Aurora PostgreSQL, and self-managed PostgreSQL.

By following this runbook, you should be able to educate the customer about key best practices, and help the customer apply the knowledge to identify PostgreSQL best practices as it pertains to schema maintenance and upkeep.

This document will be updated in response to field feedback and lessons learned. We encourage you to share your feedback, which will allow us to evolve and improve the runbook.

What is a Health-check?

A PostgreSQL Health-Check is meant to be a quick dive into a PostgreSQL database in order: 1). Check for common error patterns commonly found in PostgreSQL databases that can hinder performance, 2). Troubleshoot general performance complaints such that they can be made into actionable schema enhancements, and 3). Gather performance data/metrics at a database and query level such that specific performance issues can be troubleshooted.

Key areas to examine as a part of this procedure include:

1. Checking the database for both unused and duplicate indexes using PostgreSQL statistics.
2. Checking the database for table/index bloat (dead rows not cleaned up by vacuum) estimates

3. Evaluate overall table size and identify opportunities for table partitioning
4. Review logging levels and capture explain plans for poorly performing queries (as needed)
5. Review database settings and instance/hardware sizing in order to ensure that PostgreSQL can both utilize available system resources and that enough system resources exist to provide workload headroom for growth.

When deciding if a health check on a PostgreSQL database is necessary, one must consider the following questions:

1. Are there current complaints of general database performance from the consumers of a given database?
2. Have there been any unplanned outages, failovers, or database restarts that affect your uptime SLA/SLO?
3. Is disk usage growing exponentially without a significant increase in database activity?

The above questions are not a complete list, rather they are basic questions to start the PostgreSQL performance conversation. Worth noting, it's a best practice to review the data points we touch on below periodically even if there is no apparent issue. Also note, this health check is conducted on a per-database level (and many PostgreSQL instances contain multiple databases).

Queries used for a PostgreSQL Health-Check

The queries below are referenced in the Open-Source PostgreSQL documentation and recommended for gathering the data we'll need to perform the health-check:

```
SELECT
schemaname AS schema_name,
tablename AS table_name,
pg_size_pretty(total_bytes) AS total_size,
pg_size_pretty(table_bytes) AS table_size,
pg_size_pretty(index_bytes) AS index_size,
pg_size_pretty(COALESCE(toast_bytes, 0)) AS toast_size
FROM (
SELECT *,
total_bytes - index_bytes - COALESCE(toast_bytes, 0) AS
table_bytes
FROM (
SELECT c.oid,
nspname AS schemaname,
relname AS tablename,
pg_total_relation_size(c.oid) AS total_bytes,
pg_indexes_size(c.oid) AS index_bytes,
pg_total_relation_size(c.reltoastrelid) AS toast_bytes
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind = 'r'
AND n.nspname NOT IN ('information_schema', 'pg_catalog')
) a
) a
ORDER BY total_bytes DESC
LIMIT 20;
```

Usage: *List the 20 largest tables in your PostgreSQL database*

Query description: *Displays schema and table names, total size, table size, index size, and TOAST size.*

Sample Output:

```
schema_name | table_name        | total_size | table_size | index_size | toast_size
------------+-------------------+------------+------------+------------+-----------
-
public      | employees         | 1680 MB    | 726 MB     | 953 MB     | 0 bytes
public.     | projects          | 1093 MB    | 512 MB     | 581 MB     | 0 bytes
public.     | employee_projects | 984 MB     | 311 MB     | 672 MB     | 0 bytes
public      | departments       | 974 MB.    | 448 MB     | 526 MB     | 0 bytes
(4 rows)
```

```
SELECT
indrelid::regclass AS \"Associated Table Name\"
,array_agg(indexrelid::regclass) AS \"Duplicate Indexe Name\"
FROM pg_index
GROUP BY
indrelid
,indkey
HAVING COUNT(*) > 1;
```

Usage: *Find and list duplicate indexes and tables with which they are associated*

Query description: *Lists The associated table name and the name of each duplicate index.*

Sample Output:

```
Associated Table Name | Duplicate Indexe Name
-----------------------+-----------------------------------------------------------
-
Employees              | {idx_emp_department,idx_emp_department_dup1}
employee_projects      | {idx_emp_proj_emp_id,idx_emp_proj_emp_id_dup1}
projects               | {idx_proj_budget,idx_proj_budget_dup1}
employee_projects.     | {idx_emp_proj_proj_id,idx_emp_proj_proj_id_dup1}
employees              | {idx_emp_salary,idx_emp_salary_dup1,idx_emp_salary_dup2}
departments            | {idx_dept_name,idx_dept_name_dup1}
(6 rows)
```

```
SELECT schemaname,
relname AS table_name,
indexrelname AS index_name,
pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size,
idx_scan AS index_scans
FROM pg_stat_user_indexes ui
JOIN pg_index i ON ui.indexrelid = i.indexrelid
WHERE idx_scan < 1
AND pg_relation_size(i.indexrelid) > 10240
ORDER BY pg_relation_size(i.indexrelid) DESC;
```

Usage: *Find and list unused indexes and tables with which they are associated*

Query description: *Lists The associated table name and the name of each duplicate index.*

Sample Output:

```
schemaname | table_name        | index_name               | index_size | index_scans
-----------+--------------------+--------------------------+------------+-------------
public     | projects           | idx_proj_budget          | 258 MB     |           0
public     | employees          | idx_emp_salary           | 258 MB     |           0
public     | employees          | idx_emp_salary_dup1      | 258 MB     |           0
public     | departments        | idx_dept_name            | 231 MB     |           0
public     | departments        | idx_dept_name_dup1       | 231 MB     |           0
public     | employee_projects  | idx_emp_proj_emp_id      | 193 MB     |           0
public     | employee_projects  | idx_emp_proj_proj_id     | 193 MB     |           0
public     | employees          | idx_emp_department_dup1  | 58 MB      |           0
public     | employees          | idx_emp_department       | 58 MB      |           0
(9 rows)
```

```
WITH constants AS (
SELECT current_setting('block_size')::numeric AS bs, 23 AS hdr, 8 AS ma
),
no_stats AS (
SELECT table_schema, table_name,
n_live_tup::numeric as est_rows,
pg_table_size(relid)::numeric as table_size
FROM information_schema.columns
JOIN pg_stat_user_tables as psut
ON table_schema = psut.schemaname
AND table_name = psut.relname
LEFT OUTER JOIN pg_stats
ON table_schema = pg_stats.schemaname
AND table_name = pg_stats.tablename
AND column_name = attname
WHERE attname IS NULL
AND table_schema NOT IN ('pg_catalog', 'information_schema')
GROUP BY table_schema, table_name, relid, n_live_tup
),
null_headers AS (
SELECT
hdr+1+(sum(case when null_frac <> 0 THEN 1 else 0 END)/8) as nullhdr,
SUM((1-null_frac)*avg_width) as datawidth,
MAX(null_frac) as maxfracsum,
schemaname,
tablename,
hdr, ma, bs
FROM pg_stats CROSS JOIN constants
LEFT OUTER JOIN no_stats
ON schemaname = no_stats.table_schema
AND tablename = no_stats.table_name
WHERE schemaname NOT IN ('pg_catalog', 'information_schema')
AND no_stats.table_name IS NULL
AND EXISTS ( SELECT 1
FROM information_schema.columns
WHERE schemaname = columns.table_schema
AND tablename = columns.table_name )
GROUP BY schemaname, tablename, hdr, ma, bs
),
data_headers AS (
SELECT
ma, bs, hdr, schemaname, tablename,
(datawidth+(hdr+ma-(case when hdr%ma=0 THEN ma ELSE hdr%ma END)))::numeric AS datahdr,
(maxfracsum*(nullhdr+ma-(case when nullhdr%ma=0 THEN ma ELSE nullhdr%ma END))) AS nullhdr2
FROM null_headers
),
table_estimates AS (
SELECT schemaname, tablename, bs,
reltuples::numeric as est_rows, relpages * bs as table_bytes,
CEIL((reltuples*
(datahdr + nullhdr2 + 4 + ma -
(CASE WHEN datahdr%ma=0
THEN ma ELSE datahdr%ma END)
)/(bs-20))) * bs AS expected_bytes,
reltoastrelid
FROM data_headers
JOIN pg_class ON tablename = relname
JOIN pg_namespace ON relnamespace = pg_namespace.oid
AND schemaname = nspname
WHERE pg_class.relkind = 'r'
),
estimates_with_toast AS (
SELECT schemaname, tablename,
TRUE as can_estimate,
est_rows,
table_bytes + ( coalesce(toast.relpages, 0) * bs ) as table_bytes,
expected_bytes + ( ceil( coalesce(toast.reltuples, 0) / 4 ) * bs ) as expected_bytes
FROM table_estimates LEFT OUTER JOIN pg_class as toast
```

```
ON table_estimates.reltoastrelid = toast.oid
AND toast.relkind = 't'
),
table_estimates_plus AS (
SELECT current_database() as databasename,
schemaname, tablename, can_estimate,
est_rows,
CASE WHEN table_bytes > 0
THEN table_bytes::NUMERIC
ELSE NULL::NUMERIC END
AS table_bytes,
CASE WHEN expected_bytes > 0
THEN expected_bytes::NUMERIC
ELSE NULL::NUMERIC END
AS expected_bytes,
CASE WHEN expected_bytes > 0 AND table_bytes > 0
AND expected_bytes <= table_bytes
THEN (table_bytes - expected_bytes)::NUMERIC
ELSE 0::NUMERIC END AS bloat_bytes
FROM estimates_with_toast
UNION ALL
SELECT current_database() as databasename,
table_schema, table_name, FALSE,
est_rows, table_size,
NULL::NUMERIC, NULL::NUMERIC
FROM no_stats
),
bloat_data AS (
-- do final math calculations and formatting
select current_database() as databasename,
schemaname, tablename, can_estimate,
table_bytes, round(table_bytes/(1024^2)::NUMERIC,3) as table_mb,
expected_bytes, round(expected_bytes/(1024^2)::NUMERIC,3) as expected_mb,
round(bloat_bytes*100/table_bytes) as pct_bloat,
round(bloat_bytes/(1024::NUMERIC^2),2) as mb_bloat,
table_bytes, expected_bytes, est_rows
FROM table_estimates_plus
)
SELECT databasename, schemaname, tablename,
can_estimate,
est_rows,
pct_bloat, mb_bloat,
table_mb
FROM bloat_data
WHERE ( pct_bloat >= 50 AND mb_bloat >= 20 )
OR ( pct_bloat >= 25 AND mb_bloat >= 1000 )
ORDER BY pct_bloat DESC;
```

Usage: *Find and identify unused/redundant indexes*

Query description: *Lists the schema and table names, their related indexes, index size, and number of index scans per index.*

Sample Output:

```
databasename        | schemaname | tablename |    can_estimate | est_rows | pct_bloat | mb_bloat | table_mb
--------------------+------------+---------------+-------------+----------+-----------+----------+---------
hr_messy            |     public | departments.      | t |       3013590 |    67     | 298.03    | 447.852
hr_messy            |     public | employee_projects | t |       3016880 |    67      | 207.29   | 311.125
hr_messy            |     public | employees         | t |       3031450 |     66      | 481.20 | 724.648
hr_messy            |     public | projects          | t |       3017400 |    66        | 337.12 | 510.203
(4 rows)
```

```
WITH index_stats AS (
SELECT
current_database() AS database_name,
ns.nspname AS schema_name,
ic.relname AS index_name,
pg_size_pretty(pg_relation_size(ic.oid)) AS index_size,
pg_relation_size(ic.oid) AS index_size_bytes,
idx.indisunique AS is_unique,
idx.indisprimary AS is_primary,
COALESCE(NULLIF(pg_stat_user_indexes.idx_tup_read, 0), 0) AS estimated_row_count,
(pg_relation_size(ic.oid)::bigint -
COALESCE(
NULLIF(pg_stat_user_indexes.idx_tup_read::bigint, 0) *
NULLIF(pg_stat_user_indexes.idx_scan::bigint, 1),
0
)::bigint) AS estimated_bloat_bytes
FROM pg_class ic
JOIN pg_namespace ns ON ic.relnamespace = ns.oid
JOIN pg_index idx ON ic.oid = idx.indexrelid
JOIN pg_stat_user_indexes ON ic.oid = pg_stat_user_indexes.indexrelid
WHERE ic.relkind = 'i'
AND ns.nspname NOT IN ('information_schema', 'pg_catalog', 'pg_toast')
),
index_bloat AS (
SELECT
database_name,
schema_name,
index_name,
CASE
WHEN index_size_bytes > estimated_bloat_bytes THEN 'y'
ELSE 'n'
END AS can_estimate_bloat,
estimated_row_count,
pg_size_pretty(GREATEST(estimated_bloat_bytes, 0)) AS index_bloat_size,
ROUND(
100 *
GREATEST(estimated_bloat_bytes::numeric, 0) / NULLIF(index_size_bytes::numeric, 0),
2
) AS index_bloat_percent,
pg_size_pretty(index_size_bytes) AS index_size
FROM index_stats
)
SELECT
database_name,
schema_name,
```

```
index_name,
can_estimate_bloat,
estimated_row_count,
index_bloat_percent,
index_bloat_size,
index_size
FROM index_bloat
WHERE can_estimate_bloat='y'
ORDER BY schema_name, index_name;
```

Usage: *Estimates PostgreSQL Index bloat*

Query description: *Queries database statistics to estimate index bloat across a PostgreSQL database*

Sample Output:

```
database | schema | idx_name | can_est_bloat | est_row_count | idx_bloat_percent | idx_bloat_size | idx_size
---------+--------+----------+---------------+---------------+-------------------+----------------+-----------
hr_messy | public | departments_pkey    | y | 1444654  |        0.00 |             0 bytes        | 64 MB
hr_messy | public | employees_pkey.     | y | 9185866  |        0.00 |             0 bytes        | 65 MB
hr_messy | public | idx_emp_proj_emp_id | y | 9058290  |.       0.00 |             0 bytes.       | 158 MB
hr_messy | public | idx_emp_proj_proj_id| y | 6041578  |        0.00 |             0 bytes        | 64 MB
hr_messy | public | idx_emp_salary_dup2 | y | 87758855 |        0.00 |             0 bytes        | 258 MB
hr_messy | public | idx_proj_budget_dup1| y | 20083754 |        0.00 |             0 bytes        | 258 MB
hr_messy | public | projects_pkey       | y | 9564994  |        0.00 |             0 bytes.       | 65 MB
(7 rows)
```

Performing a Health-Check

This is the recommended order of operations in which to conduct a PostgreSQL Health-Check. While the types of queries you may need to run for your specific purposes may vary depending on your use-case and/or specific performance complaint, this order of operations is a logical start:

1. Using the *Top 20 largest tables* query to get a list of the 20 largest tables in your database. Note the largest tables so that we can dive deeper into those later.
2. **Using the** *Display duplicate indexes* **query, obtain a list of database indexes which are duplicates of others which already exist in the database.**
3. **Using the** *Find and display unused/unscanned indexes* query, locate all indexes across the database which have never been scanned in the course of database operation, noting them for follow up.
4. Using the *Estimates PostgreSQL Table and index bloat* query, generate query output estimating the table and index bloat across the database.

What is database “bloat”?

In PostgreSQL, when a row is marked for removal due to an update/delete, it is not removed right away. Instead, the row is marked as “dead”, and the database continues to operate. Later, a process known as VACUUM (or autovacuum) eventually cleans up the “dead” rows. In many cases, new DBAs can forget that databases need to be VACUUMed periodically, or autovacuum tuning configurations may not be sufficient to VACUUM the table frequently enough to prevent “dead” rows from building up. These “dead” rows are referred to as “bloat”

Generating query EXPLAIN plans

If there are any queries of concern that need to be explored more deeply, generate an explain plan for them by adding `EXPLAIN` in front of the query. Note, that running EXPLAIN only generates an estimated execution plan. If more detail tuning needs to be performed, perform an EXPLAIN ANALYZE on the queries once identified. Do note that EXPLAIN ANALYZE actually executes the query it’s used with, use caution when deploying EXPLAIN ANALYZE at scale (especially for INSERT/UPDATE/DELETE queries).

Below is a basic example of how EXPLAIN (which generates an estimated explain plan) can be achieved on a one-off basis:

```
explain select *
from pgbench_accounts
where bid = 1;
```

While this approach can be useful if we know what the problematic queries are, it’s sometimes advisable to automatically generate EXPLAIN plans such that they can be automatically inserted into the postgres logfile for later troubleshooting. This can be accomplished using the auto_explain extension (also available in RDS and Aurora PostgreSQL.

In order to enable auto_explain, first add the extension to the shared_preload_libraries setting in your postgresql.conf file or cluster/instance parameter group if using AWS managed PostgreSQL. Enabling this extension does require a database restart, which should be planned around maintenance windows if possible.

While using auto_explain to log every query is possible, it can add (sometimes significant) overhead to your database. Usually the overhead is low, though this should be tested in dev/test environments before deploying to production workloads. Overhead can be lessened by also setting the auto_explain.log_min_duration parameter in you postgresql.conf or cluster/instance parameter group (for RDS and Aurora Postgres) in order to capture execution plans for specific desired queries (regardless of execution time). Assuming the extension has already been enabled, this can be accomplished as follows:

```
LOAD 'auto_explain';

SET auto_explain.log_min_duration = 1;

SET auto_explain.log_analyze = true;

SELECT *
      FROM pgbench_accounts
WHERE bid = 1;
```

Acting on the Health-Check Data

Once the health-check itself has been completed, we can now examine the data collected and decide how to act on it. Keep in mind, performance statistics and indexes are a *per instance* resource (unless you're using Aurora PostgreSQL), and indexes not in use on writer instances may be in use on readers for specific purposes.

Unused/Duplicate Indexes

While indexes that appear to have never been scanned can usually be removed safely to remove overhead from their underlying tables, one must consider – how accurate are those statistics and when was the last time they were reset? Usually, database statistics have to be reset manually, but in some cases database statistics can be removed automatically in the event of a database crash (unclean restart of the database processes). Also worth considering, if specific tables/indexes had VACUUM successfully execute against them for days/weeks/months, the database statistics we're examining in this process may be out of date. Database bloat estimates can be used to rule out the latter, since low levels of bloat infer that tables/indexes are being appropriately vacuumed (and that database statistics are fairly accurate).

Concerning duplicate indexes, it's always a best practice to check with Application/Database developers that duplicate indexes do not have another. In some cases, they can be used for specific one-off queries in specific circumstances. Even if the scan count is 0, these unused indexes could have been created for upcoming application features that are not yet in use (and will be needed later). In many cases, duplicate indexes have been created by accident, but it goes without saying DO NOT DROP SCHEMA OBJECTS WITHOUT CHECKING WITH DEVELOPERS!

Taming larger tables

When unpartitioned (monolithic) tables grow past several 100 million rows (and/or not able to fit it's working set into available system memory), table size can play cause challenges for database performance. Usually, a handful of tables will tend to grow faster than others, especially as workloads rapidly grow. This can cause a variety of issues for database performance, including increased query run time, query locking issues, longer bulk loads, and longer times to recreate indexes. Many of these concerns (when due to table size) can be addressed by partitioning the table into a series of of smaller (linked) tables using a partitioning key. More information can be found at this blog link

Improve performance and manageability of large PostgreSQL tables by migrating to partitioned tables on Amazon Aurora and Amazon RDS

Addressing Table Bloat

Table and index bloat are problematic in a PostgreSQL database for the following reasons: 1). The excess dead rows causing table bloat result in increased billing for database storage/disk usage 2). Increased query execution time due to excessive dead rows being scanned at query execution 3). Increased number of physical I/Os/inefficient memory usage and 4). Fixing the issue with pg_repack requires additional disk allocation to accomplish successfully.

Database table/index bloat can be the root cause of performance problems in queries that previously performed well at the same workload scale. Please see the blog link below for more information on how to find/fix table bloat using the PostgreSQL extension pg_repack:

Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack

What does an EXPLAIN plan tell us

While the output from EXPLAIN and EXPLAIN ANALYZE can be difficult to read for the uninitiated, there are tools that can be used in order to make query output more human-readable. https://explain.dalibo.com/ and other tools like it can be a valuable tool for visualizing your query explain plans in order to see the highest-cost components of execution. While sometimes queries can be tuned using the pg_hints_plan extension to force query execution in a specific way, often times query explain plans will show us areas of inefficiency that will require rewrites of specific parts of the query at hand. This may mean decomposing JOIN operations into their individual pieces for efficiency, diagnosing and correcting the incorrect usage of specific PostgreSQL data types, and in some cases exposing issues relating to database bloat and overall table size/scale (discussed above). In other cases, some queries need to be re-architected to be more specific, such as adding a WHERE clause to queries in the style of `SELECT * FROM TABLE`. For specific advice on tuning PostgreSQL queries, reach out to Specialist Solutions Architects focusing on PostgreSQL. How to get them involved can be found below in the Resources section.

For more information on tuning PostgreSQL queries using both EXPLAIN plans and AWS tools, please see the blog link below:
Optimizing and tuning queries in Amazon RDS PostgreSQL based on native and external tools

Best Practices

Review monitoring/alerting strategy

If using RDS or Aurora PostgreSQL, ensure that Performance Insights and Cloudwatch enhanced monitoring are enabled at appropriate levels of data retention. While retaining 1-2 weeks worth of data is usually sufficient for troubleshooting one-off issues, keeping one to two financial quarters worth of data helps with troubleshooting issues related to workload growth/scale vs the scale at some point in the past. Ensure all alerts have appropriate thresholds set – too low and they'll tend to get ignored, to high and it limits time to respond to potential problems.

If using self-managed PostgreSQL, pgBadger can be an excellent option for visualizing PostgreSQL logs to review for anomalies in performance

Ensure regular maintenance tasks are completed regularly

While AUTOVACUUM and AUTOANALYZE are enabled by default in most PostgreSQL configurations, performing regular VACUUM ANALYSE operations on a weekly or monthly basis across the database cluster can help manage table bloat and ensure database statistics are kept up-to-date. While these operations can be scheduled using cron on a database using pg_cron or via Lambda/Step Function, it is vital to ensure that production levels of monitoring, logging, and alerting are enabled if these mechanism are meant to be relied upon instead of regular human intervention.

Review logs and metrics on a schedule

While we usually rely on alerts to tell us when something is going wrong or trending in the wrong direction with our PostgreSQL database, regular review of PostgreSQL logs and metrics can help spot both issues related to workload (failed data loads, error, fatal, warnings). Whether relying exclusively on PostgreSQL logs/visualizations or you have a more comprehensive solution implemented (such as Cloudwatch Enhanced metrics and Performance Insights), reviewing the contents of these data sources on a regular basis can be a valuable tool in finding and diagnosing minor database issues before they grow into much bigger issues that require immediate intervention.

Configuration management

If using AWS managed PostgreSQL (Aurora or RDS Postgres), configuration management is simpler then with self-managed. In this case, PostgreSQL parameters are managed via cluster/instance parameter groups (Aurora_, and database parameter groups (RDS Postgres). It is a best practice to create custom parameter groups for each database/cluster you're managing, ideally created from the default templates we provide for each PostgreSQL major version. Use caution when changing any parameter that is "Formula" based, meaning that the value changes based on instance size/class used with that parameter group. Otherwise, changes to the mentioned parameter groups can be tracked using CloudTrail.

If using self-managed PostgreSQL, configuration management is still possible and easy to implement using GIT repositories. Using the managed GIT service (or local GIT server of your choice), create a repository within your PostgreSQL database directory (only including specific database configuration files you desire to manage), and configuration updates can be as simple as periodic `GIT PULL` operations scheduled locally via `cron`.

For more information on configuration management for AWS Managed PostgreSQL, please see the link below:
[Working with parameters on your RDS for PostgreSQL DB instance]

Resources

Public knowledge resources

- PostgreSQL Health-Check code repository: Sample Health-Check Resources for PostgreSQL
- Documentation: Open-Source PostgreSQL documentation
- Documentation: Aurora/RDS Postgres Documentation

Training/Hands-On Workshops

- Training: Aurora PostgreSQL Immersion Day
- Training: RDS PostgreSQL Immersion Day
- Training: Troubleshoot Amazon Aurora PostgreSQL Performance Workshop
- Training: PostgreSQL Fundamentals

Getting help troubleshooting PostgreSQL workloads

- Use Premium Support engagements (Support Cases) for service troubleshooting and break & fix assistance.
- Internal wiki supported by Specialist Solutions Architects for PostgreSQL
- Use SpecReqs for architectural guidance, and assistance with troubleshooting customer PostgreSQL performance concerns. Note that all SpecReqs must be created as “Sales Support”, and must be linked to an SFDC opportunity with an estimated non-zero opportunity amount. Requests that don’t meet those requirements may not be fulfilled.

Contributors

The following individuals and organizations contributed to this document:

- Peter Celentano - Senor DB Specialist Solutions Architect, Amazon Web Services

Document history

Change	Description	Date
Initial publication	Initial publication	July 2024