Amazon Aurora Performance Optimization Techniques

Rajesh Matkar & Arabinda Pani
Prin. Database Specialist Solutions Architects

Agenda

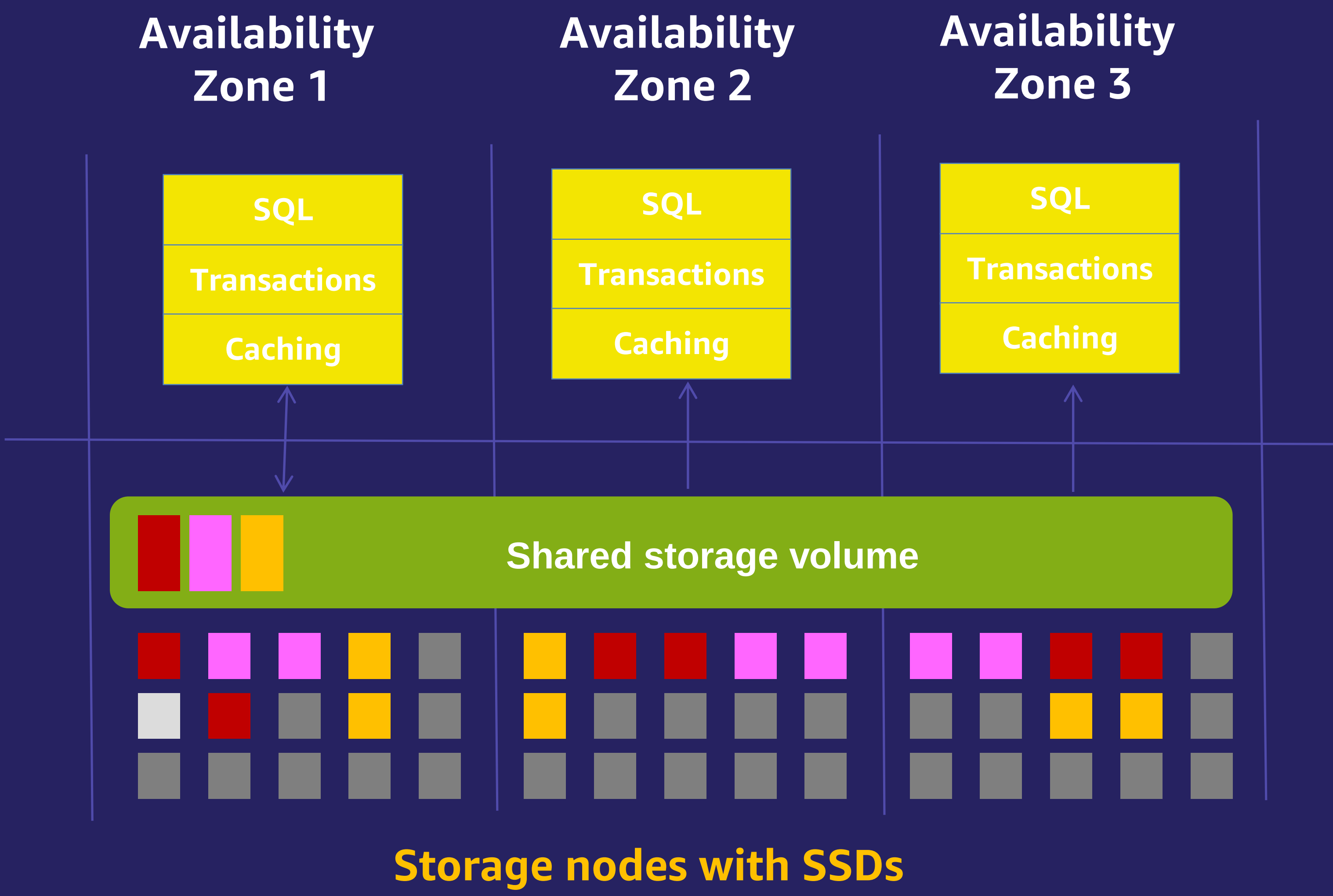
- Amazon Aurora architecture
- Root cause vs. symptoms
- Database monitoring services
- Monitoring Aurora MySQL and Query Tuning
- Monitoring Aurora PostgreSQL, Optimizing and Query Tuning
- Partner Packages
- Q & A

Amazon Aurora Architecture

How is Aurora different?

Amazon Aurora – Leverages a scale-out, distributed architecture

- Purpose-built log-structured distributed storage system designed for databases
- Storage volume is striped across hundreds of storage nodes distributed over 3 different availability zones
- Six copies of data, two copies in each availability zone to protect against AZ+1 failures
- 10GB of segment size
- Master and replicas (up to 15) all point to the same storage



Amazon Aurora Performance enhancements

5x better throughput than standard MySQL and 3x better throughput than standard PostgreSQL

Amazon Aurora

- Only writes log records to the storage
- Log buffer removed
- Checkpoints and Full page writes removed
- Survivable Cache
- Asynchronous DB backup at Aurora storage layer

Aurora MySQL

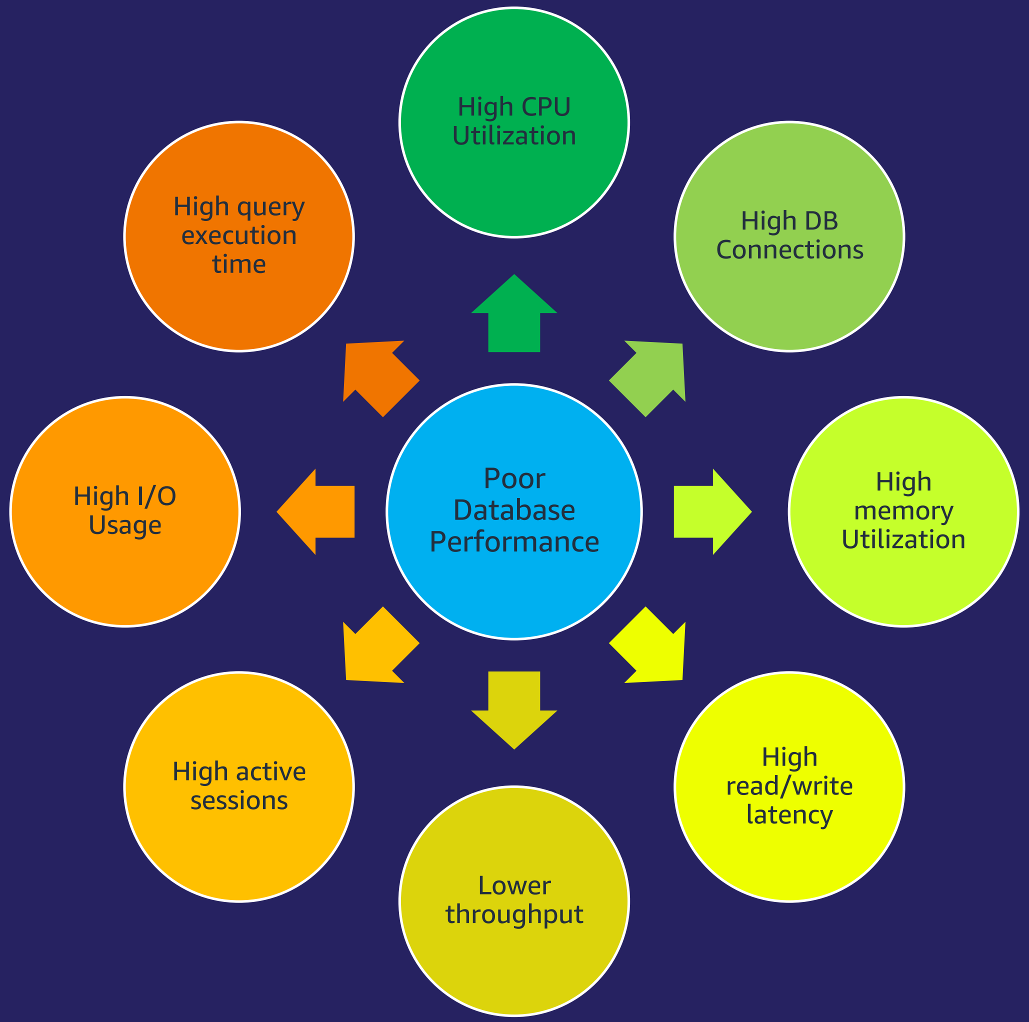
- Asynchronous group commit and key prefetch
- Latch-free lock manager
- Fast B-Tree inserts
- Logical read ahead
- NUMA aware and Smart thread scheduler
- High-performance auditing
- Parallel Query

Aurora PostgreSQL

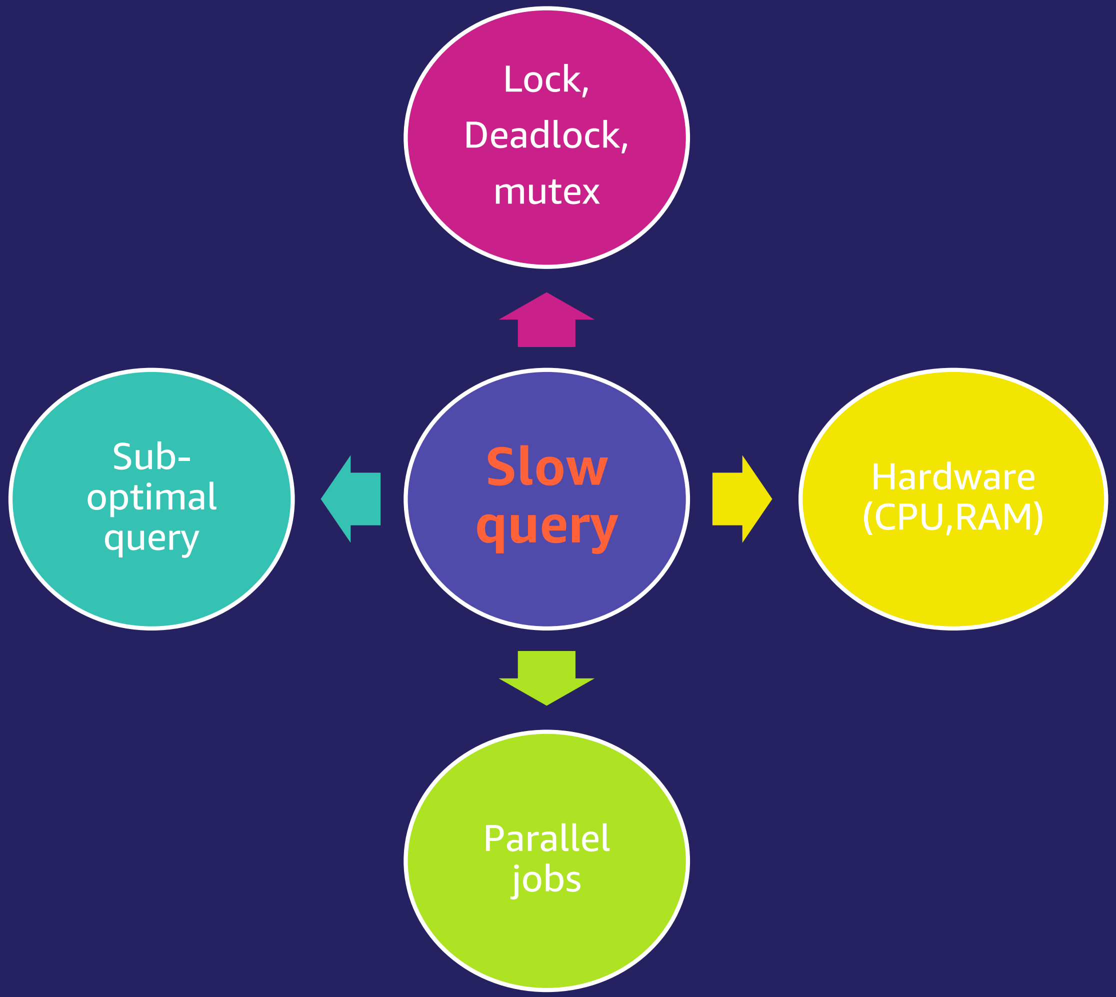
- No double buffering
- Intelligent Prefetch for index and index-only range scan
- Intelligent Vacuum Prefetch
- Cluster Cache Management (CCM)
- Query Plan Management (QPM)

Chasing root cause and symptoms

Difference between root cause and symptoms

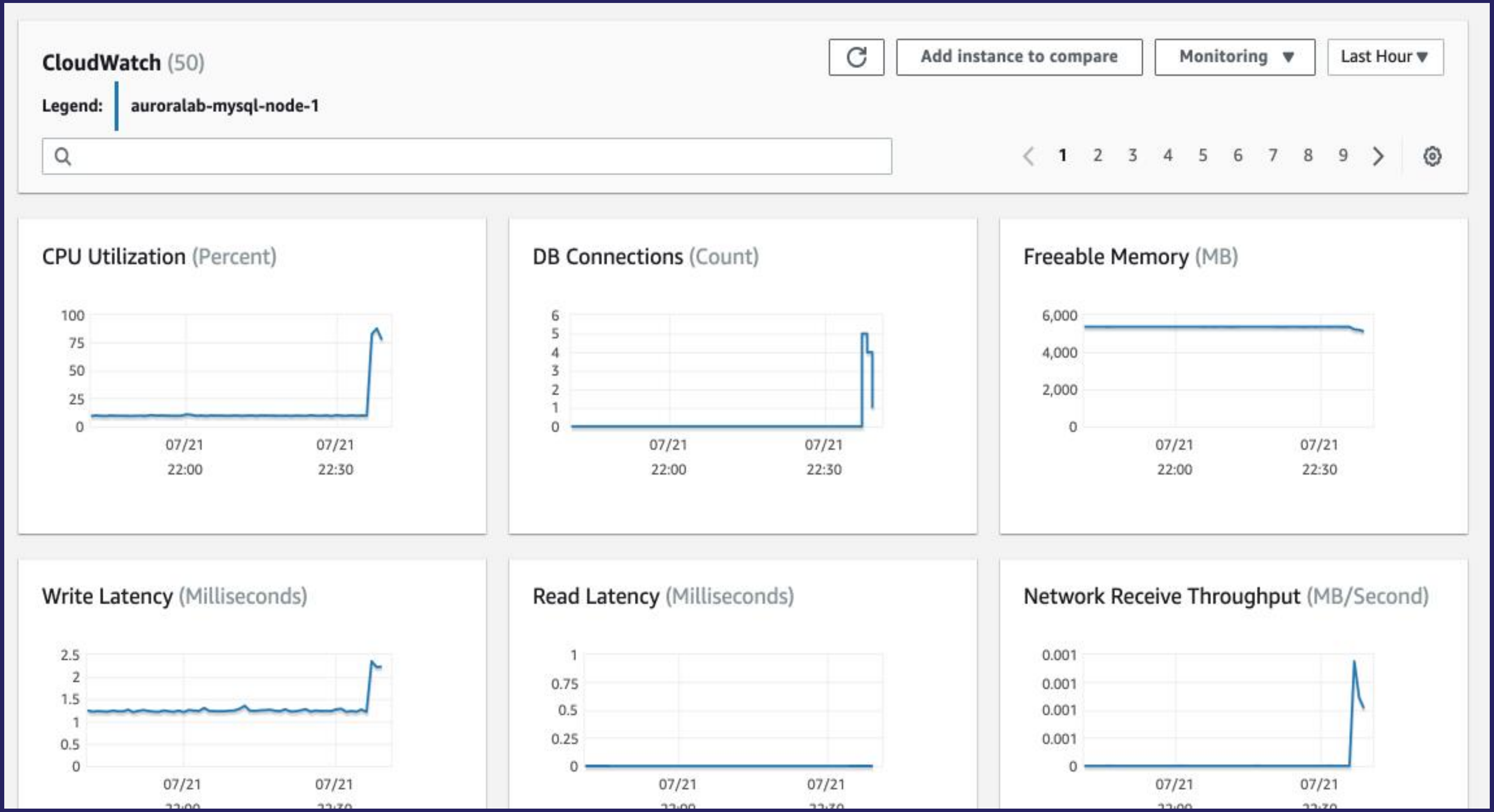


What can cause a slow query



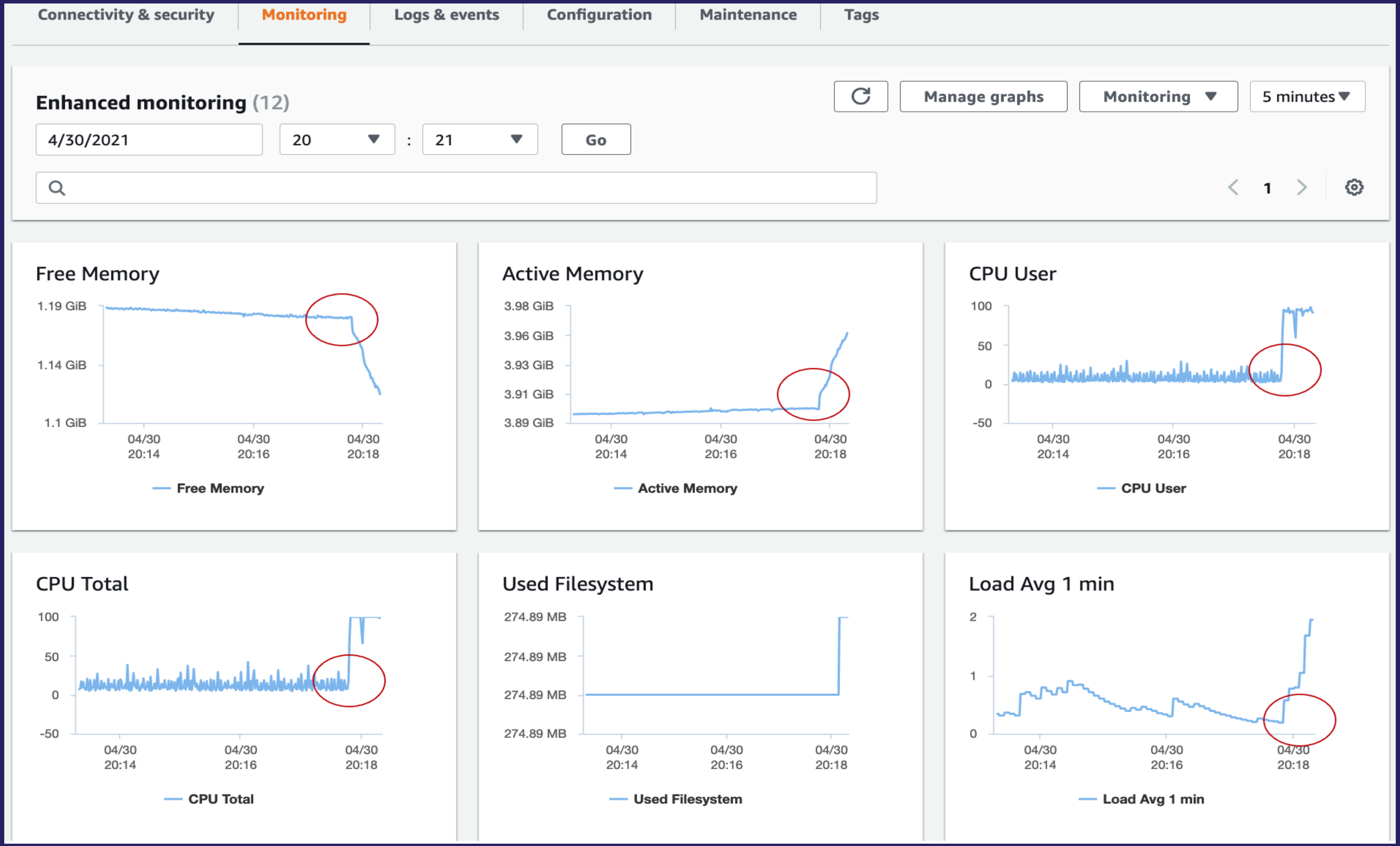
Database Monitoring Services

CloudWatch Metrics

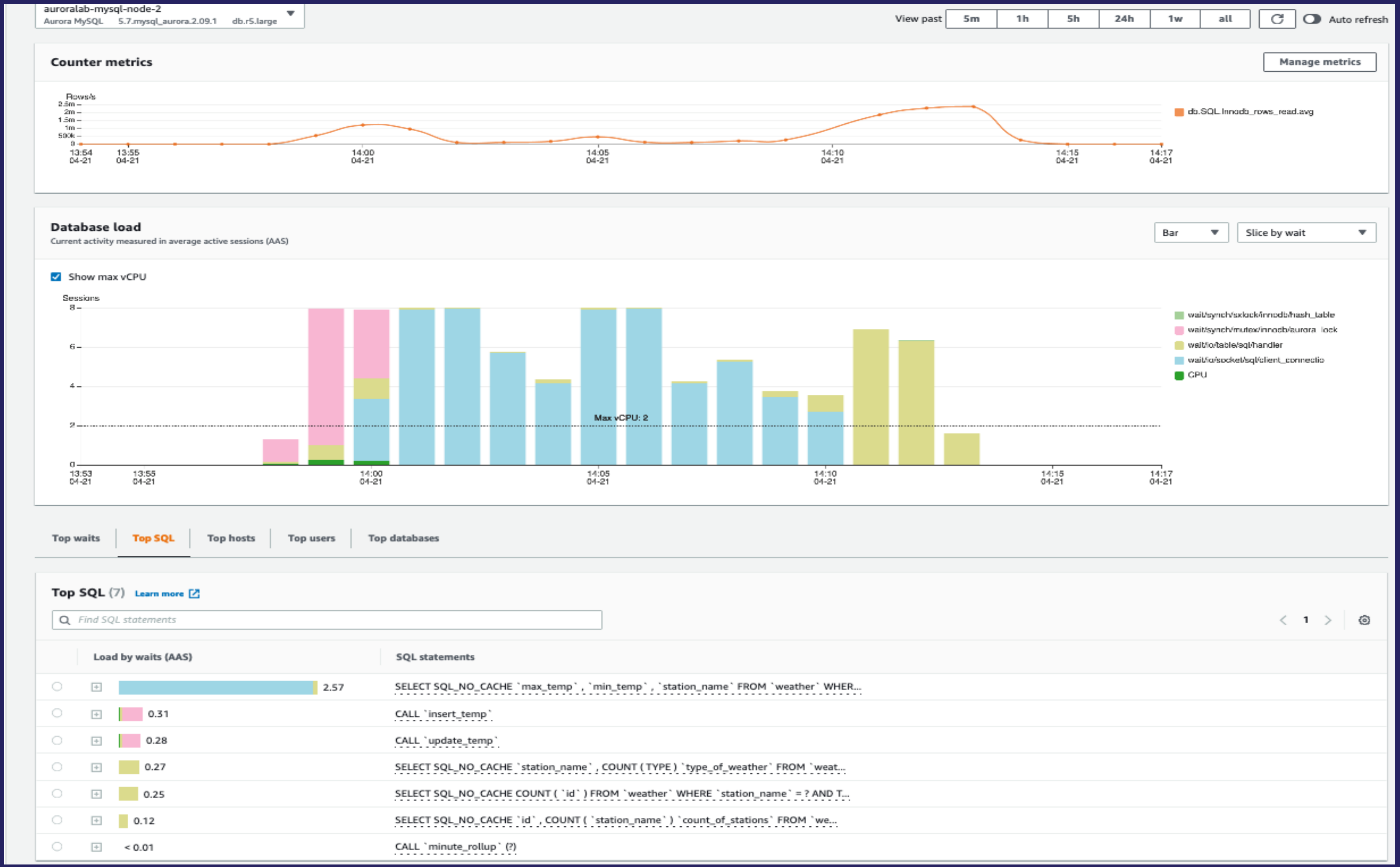


CW alarms can be created for important metrics

Enhanced monitoring – Viewing Operating System metrics



RDS Performance Insights

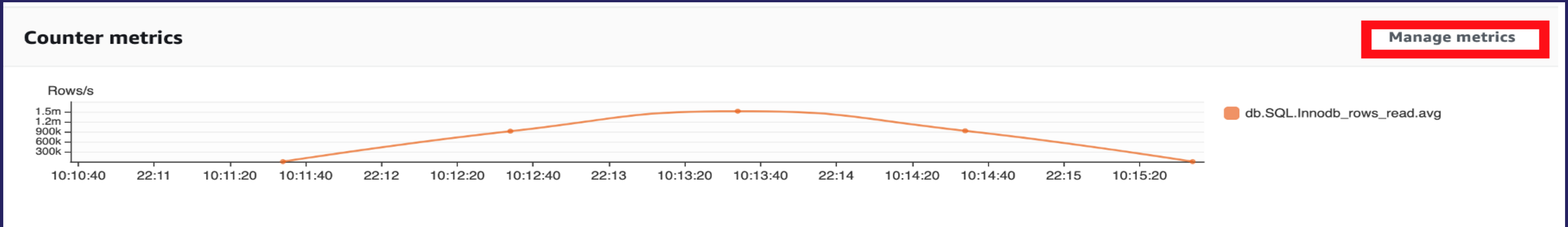


Counter Metrics

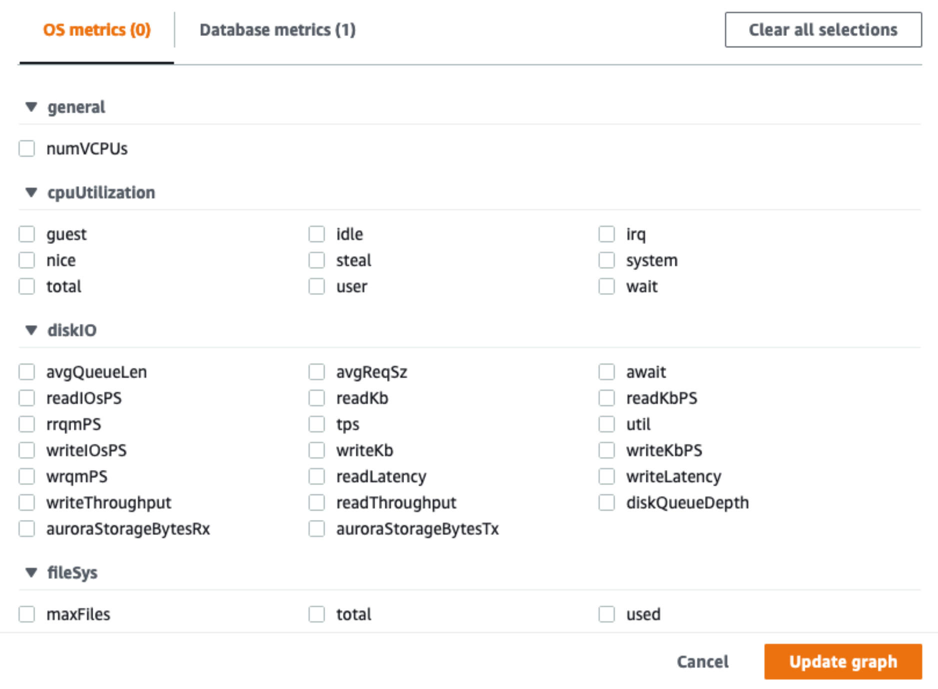
Database Load

Top SQL Activities

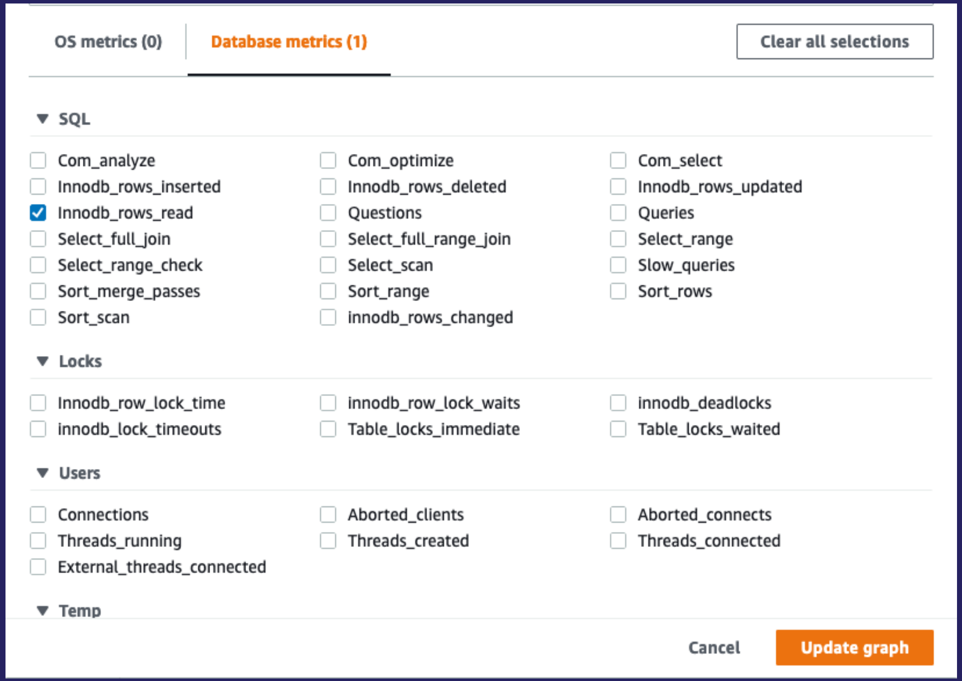
RDS Performance Insights – Counter metrics



OS counter metrics

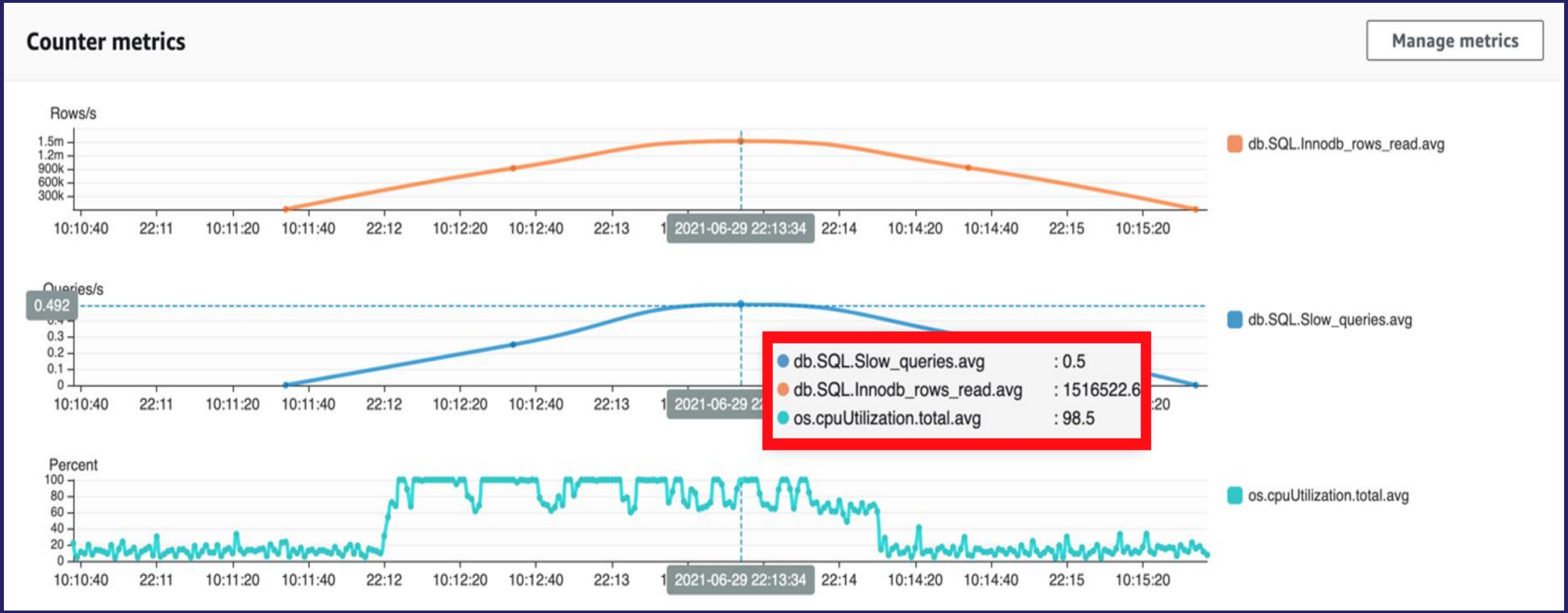


Database counter metrics



RDS Performance Insights

Example after adding OS & Database counter metrics



RDS Performance Insights

Default SQL query browser view

RDS Performance Insights

Sample view after adding custom metrics

Top waits | **Top SQL** | Top hosts | Top users | Top databases

Top SQL (6) Learn more

Find SQL statements

Load by waits (AAS)	SQL statements	Calls/sec	Rows examined/sec	Rows sent/sec	Avg latency (ms)/c...	Rows examined/call	Rows sent/call
1.06	UPDATE `mylab` . `weather` SET `max_temp` = ? WHERE `id` = ?	0.06	176846.08	0.00	15362.48	3196833.04	0.00
0.51	CALL `insert_temp`	0.03	0.00	0.00	13770.89	0.00	0.00
0.51	DELETE from mylab.weather where serialid=key_value	-	-	-	-	-	-
0.21	SELECT SQL_NO_CACHE COUNT (`id`) FROM `weather` WHERE `station_name` = ? AND T...	0.05	149638.99	0.05	3178.32	3196832.91	1.00
0.13	SELECT SQL_NO_CACHE `max_temp` , `min_temp` , `station_name` FROM `weather` WHER...	0.02	61233.55	17.60	2856.36	3197752.11	919.11
< 0.01	SELECT SQL_NO_CACHE `k` , COUNT (`k`) , `SQRT` (SUM (`k`)) , `SQRT` (AVG ...	0.03	61.48	20.44	15.91	1806.00	600.44
< 0.01	CALL `minute_rollup` (?)	0.00	0.00	0.00	-	-	-

SQL information

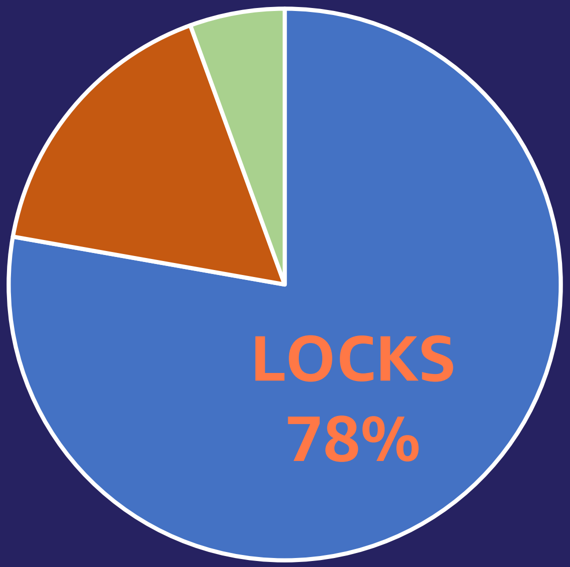
If the SQL statement exceeds 4096 characters, it is truncated. To view the full SQL statement, choose **Download**.

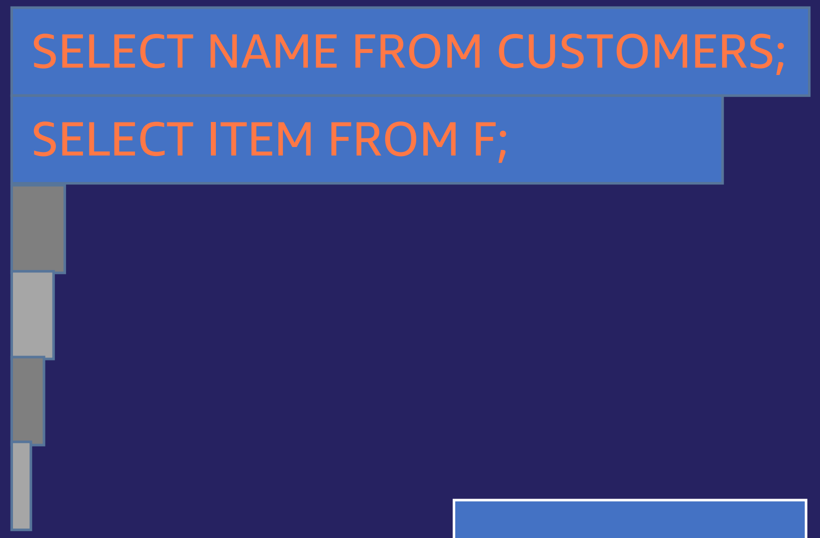
```
DELETE from mylab.weather  where serialid=key_value
```

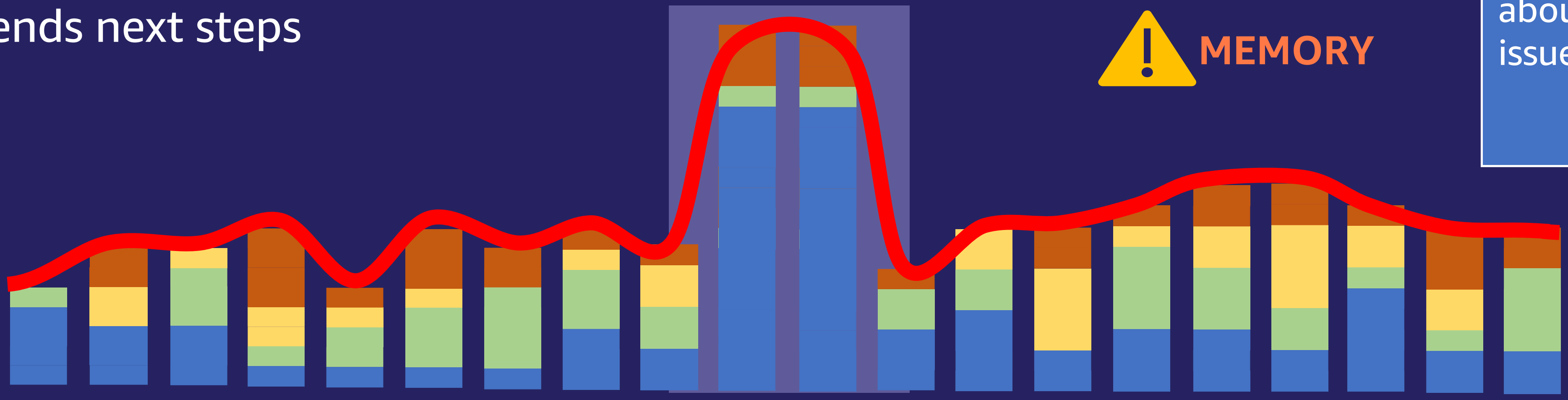
DevOps Guru for RDS

New!

- **Finds DB performance anomalies**
- **Analyzes the anomaly**
- **Highlights:**
  - Prevalent wait events
  - Prevalent SQL statements
  - Other anomalous metrics
- Recommends next steps







Monitoring Aurora MySQL

Monitoring options for Aurora MySQL

MySQL Engine

- General Logs
- Slow query logs
- Processlist
- InnoDB Monitor
- Global Status
- Performance Schema
- Sys Schema
- Information_schema.Innodb_metrics

Aurora

- CloudWatch Metrics
- Enhanced Monitoring
- Performance Insights
- CloudWatch Log Insights
- DevOps Guru for RDS

Query Analysis

- Explain
- Profile
- Performance schema
- Optimizer trace

Identify slow queries using MySQL slow query log

Log queries based on pre-defined execution time and rows processing limits.
Find queries which are taking longer time to execute and target for optimization

Query_time : The statement time in seconds.
Lock_time : The time to acquire locks in seconds.
Rows_sent : The number of rows sent to client.
Rows _examined : The number of rows examined by the server layer (not counting processing internal to storage engines).

```
Time Id Command Argument
# Time: 2021-05-06T15:41:44.636025Z
# User@Host: masteruser[masteruser] @ [54.240.197.239] Id: 57
# Query_time: 5.573166 Lock_time: 0.000131 Rows_sent: 1120 Rows_examined: 3197953
use mylab;
SET timestamp=1620315704;
SELECT sql_no_cache max_temp,min_temp,station_name FROM weather WHERE max_temp > 23 and id = 'USC00103882' ORDER BY
max_temp DESC;
# Time: 2021-05-06T15:41:44.830150Z
# User@Host: masteruser[masteruser] @ [54.240.197.231] Id: 54
# Query_time: 5.637954 Lock_time: 0.000110 Rows_sent: 1 Rows_examined: 3196833
SET timestamp=1620315704;
SELECT sql_no_cache count(id) FROM weather WHERE station_name = 'EAGLE MTN' and type = 'Weak Cold';
# Time: 2021-05-06T15:41:44.891170Z
# User@Host: masteruser[masteruser] @ [54.240.197.231] Id: 55
# Query_time: 5.445422 Lock_time: 0.000130 Rows_sent: 0 Rows_examined: 3196833
SET timestamp=1620315704;
SELECT sql_no_cache max_temp,min_temp,station_name FROM weather WHERE max_temp > 28 and id = 'USC00046699' ORDER BY
max_temp DESC;
# Time: 2021-05-06T15:41:46.024487Z
# User@Host: masteruser[masteruser] @ [54.240.197.239] Id: 56
# Query_time: 6.980112 Lock_time: 0.000000 Rows_sent: 0 Rows_examined: 3196834
SET timestamp=1620315706;
CALL insert_temp;
# Time: 2021-05-06T15:41:48.829288Z
# User@Host: masteruser[masteruser] @ [54.240.197.231] Id: 53
```

Analyze slow query log file using pt-query-digest

pt-query-digest is a open source tool from Percona which analyzes MySQL queries from slow, general, and binary log files.

```
ubuntu@ip-172-31-0-244:~$ pt-query-digest slow_log.txt

# 140ms user time, 10ms system time, 29.84M rss, 36.85M vsz
# Current date: Tue May  4 18:04:45 2021
# Hostname: ip-172-31-0-244
# Files: slow_log.txt
# Overall: 113 total, 4 unique, 0.56 QPS, 4.87x concurrency ______________
# Time range: 2021-05-04T17:53:32 to 2021-05-04T17:56:52
# Attribute          total     min     max     avg     95%  stddev  median
# ============     ======= ======= ======= ======= ======= ======= =======
# Exec time           975s      2s     27s      9s     19s      7s      4s
# Lock time           287s       0     22s      3s     15s      5s   119us
# Rows sent         31.14k       0   1.20k  282.21   1.14k  489.35    0.99
# Rows examine     344.54M   3.05M   3.05M   3.05M   3.03M       0   3.03M
# Query size         9.15k      16     129   82.93  124.25   40.24   92.72

# Profile
# Rank Query ID                           Response time  Calls R/Call  V/M
# ==== ================================== ============== ===== ======= ===
#    1 0x46C4B9DF12817007A6F4BC65D4AFF61F 395.1522 40.5%    24 16.4647  1.31 UPDATE mylab.weather
#    2 0xAC8DD5BBF3975693C05247449313884D 341.0392 35.0%    23 14.8278  1.42
#    3 0x39F9DCD0C06AA3B975CAF431D0B72222 129.1875 13.3%    36  3.5885  1.23 SELECT weather
#    4 0x98D290535EAFFBBA08169665326CF519 109.1601 11.2%    30  3.6387  1.41 SELECT weather
```

```
# Query 1: 0.13 QPS, 2.11x concurrency, ID 0x46C4B9DF12817007A6F4BC65D4AFF61F at byte 3042
# This item is included in the report because it matches --limit.
# Scores: V/M = 1.31
# Time range: 2021-05-04T17:53:39 to 2021-05-04T17:56:46
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count         21      24
# Exec time     40    395s      7s     27s     16s     23s      5s     15s
# Lock time     99    287s      2s     22s     12s     17s      4s     11s
# Rows sent      0       0       0       0       0       0       0       0
# Rows examine  21  73.17M   3.05M   3.05M   3.05M   3.03M       0   3.03M
# Query size    15   1.43k      61      61      61      61       0      61
# String:
# Databases    mylab
# Hosts        172.31.0.244
# Users        masteruser
# Query_time distribution
#   1us
#  10us
# 100us
#   1ms
#  10ms
# 100ms
#    1s  #####
#  10s+  ################################################################
# Tables
#    SHOW TABLE STATUS FROM `mylab` LIKE 'weather'\G
#    SHOW CREATE TABLE `mylab`.`weather`\G
UPDATE mylab.weather SET max_temp = 44 where id='USC00103882'\G
# Converted for EXPLAIN
# EXPLAIN /*!50100 PARTITIONS*/
select  max_temp = 44 from mylab.weather where  id='USC00103882'\G
```

Slow query details

Identify slow queries using MySQL Performance Schema

Sample queries

Queries performing full table scan

```
mysql> SELECT schema_name, substr(digest_text, 1, 100) AS statement, count_star AS cnt, sum_select_scan AS full_table_scan FROM performance_schema.events_sta
tements_summary_by_digest WHERE sum_select_scan > 0 and schema_name iS NOT NULL ORDER BY sum_select_scan desc limit 5;
+-------------+-----------------------------------------------------------------------------------------------------+-----+-----------------+
| schema_name | statement                                                                                           | cnt | full_table_scan |
+-------------+-----------------------------------------------------------------------------------------------------+-----+-----------------+
| mylab       | SELECT SQL_NO_CACHE COUNT ( `id` ) FROM `weather` WHERE `station_name` = ? AND TYPE = ?             |  25 |              25 |
| mylab       | SELECT SQL_NO_CACHE `max_temp` , `min_temp` , `station_name` FROM `weather` WHERE `max_temp` > ? AND |  22 |              22 |
| mylab       | SHOW TABLES                                                                                         |   4 |               4 |
| mylab       | SHOW SCHEMAS                                                                                        |   3 |               3 |
| mylab       | SELECT `object_schema` AS `table_schema` , `object_name` AS TABLE_NAME , `index_name` , `count_star` |   3 |               3 |
+-------------+-----------------------------------------------------------------------------------------------------+-----+-----------------+
5 rows in set (0.00 sec)
```

Top 5 wait events

```
mysql> select event_name as wait_event, count_star as all_occurrences, CONCAT(ROUND(sum_timer_wait / 1000000000000, 2), ' s') as total_wait_time, CONCAT(ROUND(avg_timer_wai
t / 1000000000000, 2), ' s') as avg_wait_time from performance_schema.events_waits_summary_global_by_event_name where count_star > 0 and event_name <> 'idle' order by sum_t
imer_wait desc limit 5;
+------------------------------------------------------+-----------------+-----------------+---------------+
| wait_event                                           | all_occurrences | total_wait_time | avg_wait_time |
+------------------------------------------------------+-----------------+-----------------+---------------+
| wait/synch/cond/sql/FILE_AS_TABLE::cond_request      |              24 | 6840.81 s       | 285.03 s      |
| wait/io/table/sql/handler                            |       341413475 | 938.86 s        | 0.00 s        |
| wait/synch/mutex/innodb/aurora_lock_thread_slot_futex |             52 | 512.47 s        | 9.86 s        |
| wait/synch/mutex/innodb/trx_mutex                    |       191613034 | 8.92 s          | 0.00 s        |
| wait/synch/sxlock/innodb/hash_table_locks            |        38316334 | 2.46 s          | 0.00 s        |
+------------------------------------------------------+-----------------+-----------------+---------------+
5 rows in set (0.02 sec)
```

Query Tuning in Aurora MySQL

Analyze slow queries using Explain Plan

Sample plan before index

```
mysql> EXPLAIN SELECT sql_no_cache max_temp,min_temp,station_name FROM weather WHERE max_temp > 42 and id = 'USC00103882' ORDER BY max_temp DESC;
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+----------+-----------------------------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra                       |
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+----------+-----------------------------+
|  1 | SIMPLE      | weather | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 3162938 |     3.33 | Using where; Using filesort |
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+----------+-----------------------------+
1 row in set, 1 warning (0.00 sec)
```

Sample plan after index

```
mysql> EXPLAIN SELECT sql_no_cache max_temp,min_temp,station_name FROM weather WHERE max_temp > 42 and id = 'USC00103882' ORDER BY max_temp DESC;
+----+-------------+---------+------------+------+---------------+--------+---------+-------+------+----------+----------------------------------------------------+
| id | select_type | table   | partitions | type | possible_keys | key    | key_len | ref   | rows | filtered | Extra                                              |
+----+-------------+---------+------------+------+---------------+--------+---------+-------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | weather | NULL       | ref  | idx_id        | idx_id | 13      | const | 1120 |    33.33 | Using index condition; Using where; Using filesort |
+----+-------------+---------+------------+------+---------------+--------+---------+-------+------+----------+----------------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

Column	Meaning
select_type	The SELECT type
type	The join type
possible_keys	The possible indexes to choose
key	The index actually chosen
key_len	The length of the chosen key
ref	The columns compared to the index
rows	Estimate of rows to be examined
filtered	Percentage of rows filtered by table condition
Extra	Additional information

Simple -> Simple SELECT (not using UNION or subqueries)

Using filesort -> If a sort can't be performed from an index, it's a filesort

Analyze slow queries using PROFILING

Sample profiling for a query without an index

```
mysql> SET profiling = 1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SELECT sql_no_cache count(id) FROM weather WHERE station_name = 'EAGLE MTN' and type = 'Weak Cold';
+-----------+
| count(id) |
+-----------+
|       348 |
+-----------+
1 row in set (1.49 sec)

mysql> SHOW PROFILES;
+----------+------------+-----------------------------------------------------------------------------------------------------+
| Query_ID | Duration   | Query                                                                                               |
+----------+------------+-----------------------------------------------------------------------------------------------------+
|        1 | 1.49353600 | SELECT sql_no_cache count(id) FROM weather WHERE station_name = 'EAGLE MTN' and type = 'Weak Cold' |
+----------+------------+-----------------------------------------------------------------------------------------------------+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW PROFILE FOR QUERY 1;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000077 |
| checking permissions | 0.000007 |
| Opening tables       | 0.000017 |
| init                 | 0.000028 |
| System lock          | 0.000008 |
| optimizing           | 0.000014 |
| statistics           | 0.000016 |
| preparing            | 0.000018 |
| executing            | [illegible] |
| Sending data         | 1.493168 |
| end                  | [illegible] |
| query end            | 0.000012 |
| closing tables       | 0.000013 |
| freeing items        | 0.000057 |
| cleaned up           | 0.000007 |
| logging slow query   | 0.000055 |
| cleaning up          | 0.000018 |
+----------------------+----------+
17 rows in set, 1 warning (0.00 sec)

mysql> SET profiling = 0;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

Sending data*

The thread is reading and processing rows for a SELECT statement, and sending data to the client. Because operations occurring during this state tend to perform large amounts of disk access (reads), it is often the longest-running state over the lifetime of a given query.

Profiling using performance_schema

Sample profiling for a query with an index

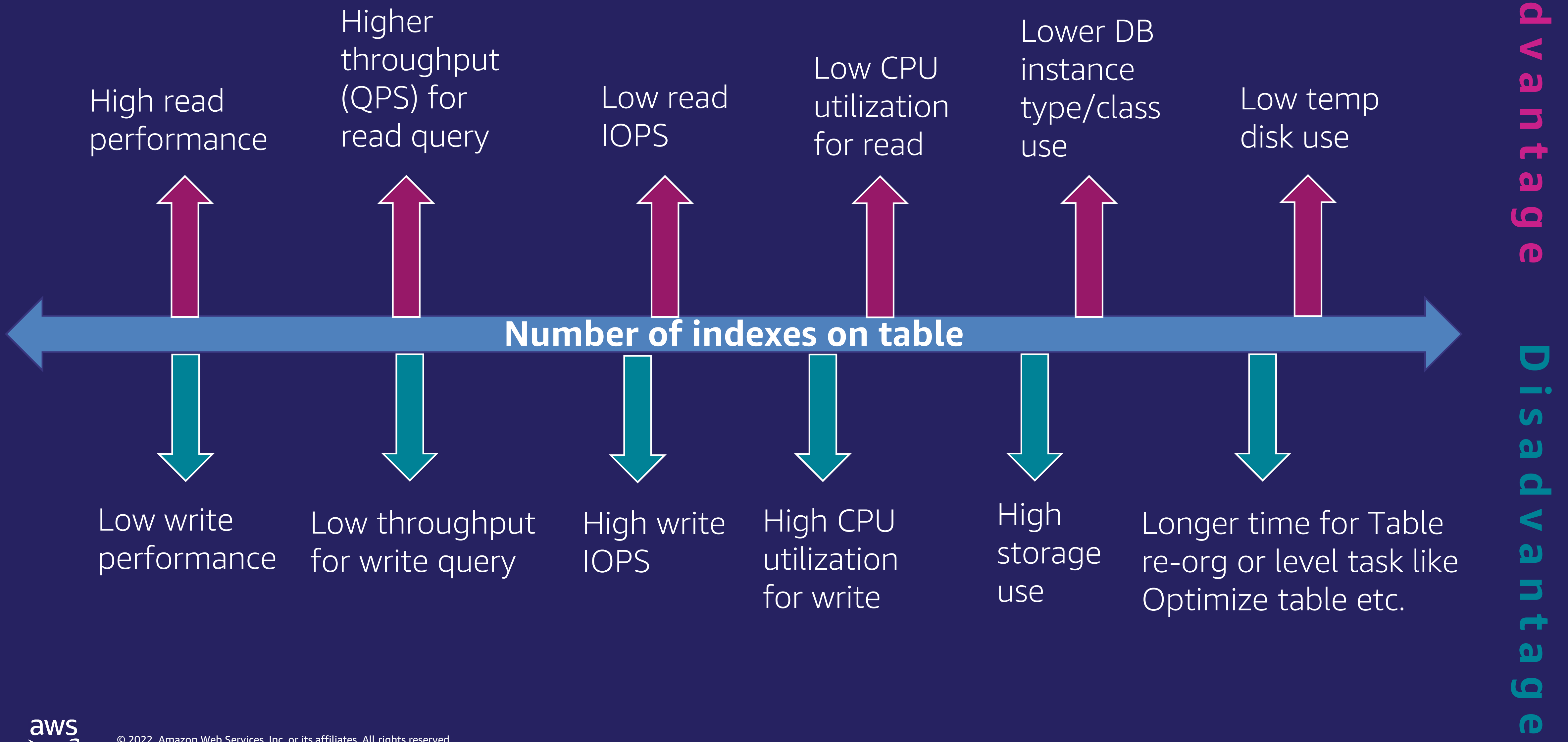
```
mysql> SELECT sql_no_cache count(id) FROM weather WHERE station_name = 'EAGLE MTN' and type = 'Weak Cold';
+-----------+
| count(id) |
+-----------+
|       348 |
+-----------+
1 row in set (0.00 sec)

mysql> SELECT EVENT_ID, TRUNCATE(TIMER_WAIT/1000000000000,6) as Duration, SQL_TEXT FROM performance_schema.events_statements_history_long WHE
RE SQL_TEXT like '%EAGLE MTN%';
+----------+----------+---------------------------------------------------------------------------------------------------+
| EVENT_ID | Duration | SQL_TEXT                                                                                          |
+----------+----------+---------------------------------------------------------------------------------------------------+
|   582117 | 0.001428 | SELECT sql_no_cache count(id) FROM weather WHERE station_name = 'EAGLE MTN' and type = 'Weak Cold' |
+----------+----------+---------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT event_name AS Stage, TRUNCATE(TIMER_WAIT/1000000000000,6) AS Duration FROM performance_schema.events_stages_history_long WHERE
NESTING_EVENT_ID=582117;
+--------------------------------+----------+
| Stage                          | Duration |
+--------------------------------+----------+
| stage/sql/starting             | 0.000067 |
| stage/sql/checking permissions | 0.000004 |
| stage/sql/Opening tables       | 0.000015 |
| stage/sql/init                 | 0.000023 |
| stage/sql/System lock          | 0.000004 |
| stage/sql/optimizing           | 0.000010 |
| stage/sql/statistics           | 0.000075 |
| stage/sql/preparing            | 0.000012 |
| stage/sql/executing            | 0.000000 |
| stage/sql/Sending data         | 0.001168 |
| stage/sql/end                  | 0.000001 |
| stage/sql/query end            | 0.000005 |
| stage/sql/closing tables       | 0.000005 |
| stage/sql/freeing items        | 0.000029 |
| stage/sql/cleaned up           | 0.000001 |
| stage/sql/cleaning up          | 0.000000 |
+--------------------------------+----------+
16 rows in set (0.01 sec)
```

Index usages for Query performance optimization

- rows filter
- avoid temporary tables use
- avoid sort operation use
- avoid reading rows from the tables (covering index)

and much more

Many indexes on table – good or bad?



Things that may go wrong

Optimizer can choose wrong index

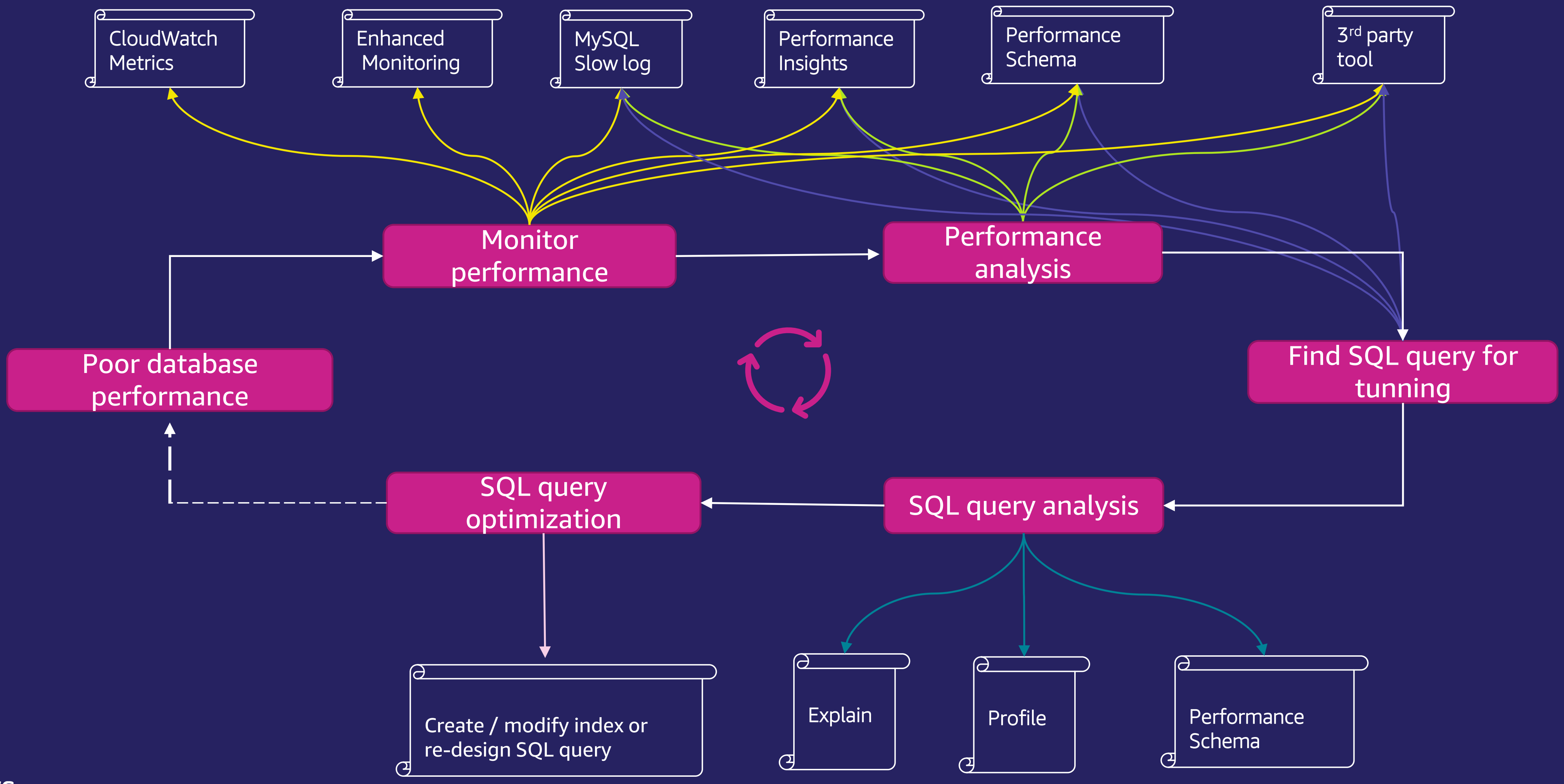
- index scan is expensive
- statistics are outdated
- innodb_stats_persistent_sample_pages
- bad queries

How to fix/workaround it

- Force index hint
- Increase innodb_stats_persistent_sample_pages*
- Analyze
- Optimize
- Move some logic to the application

*Aurora has default pages set to 256 unlike MySQL which is 8

Query Tuning Cycle



Monitoring Aurora PostgreSQL

Logging in Aurora PostgreSQL

- Aurora PostgreSQL logging
  postgresql.log.%Y-%m-%d-%H%M

- Publish PostgreSQL logs to CloudWatch Logs, perform real-time analysis using CloudWatch Log Insights and use CloudWatch to create alarms and view metrics

- Use log_fdw extension to query the PostgreSQL log via SQL for PANIC, other errors or information.

Number of parameters to control the logging

```
log_statement
log_connections
log_disconnections
log_lock_waits
Log_temp_files
log_min_duration_statement
log_autovacuum_min_duration
rds.force_autovacuum_logging_level
track_functions
log_statement_stats
pgaudit.log
auto_explain.log_min_duration
auto_explain.log_verbose
auto_explain.log_nested_statements
rds.log_retention_period
```

And many more …

PostgreSQL Extensions for Performance Monitoring

pg_stat_statements for tracking execution statistics of SQL statements

auto_explain for logging execution plans of slow queries automatically

pg_proctab exposes OS/proc information through SQL

plprofiler to find bottleneck in PL/pgSQL function and stored procedures

Other useful tools and scripts for Monitoring

pg-collector collects database information and presents it in a consolidated HTML file

PGPerfStatsSnapper for periodic collection (snapping) of PostgreSQL performance related statistics and metrics

rds-support-tools contains collection of useful database monitoring scripts

Amazon Aurora Postgres Advanced Monitoring creates CloudWatch dashboard with useful database monitoring metrics

Pgbadger PostgreSQL log analyzer with fully detailed reports and graphs

Optimizing Aurora PostgreSQL

PostgreSQL Extensions for Database and SQL Tuning

pg_repack for rebuilding a table online

pg_partman to partition tables with less effort

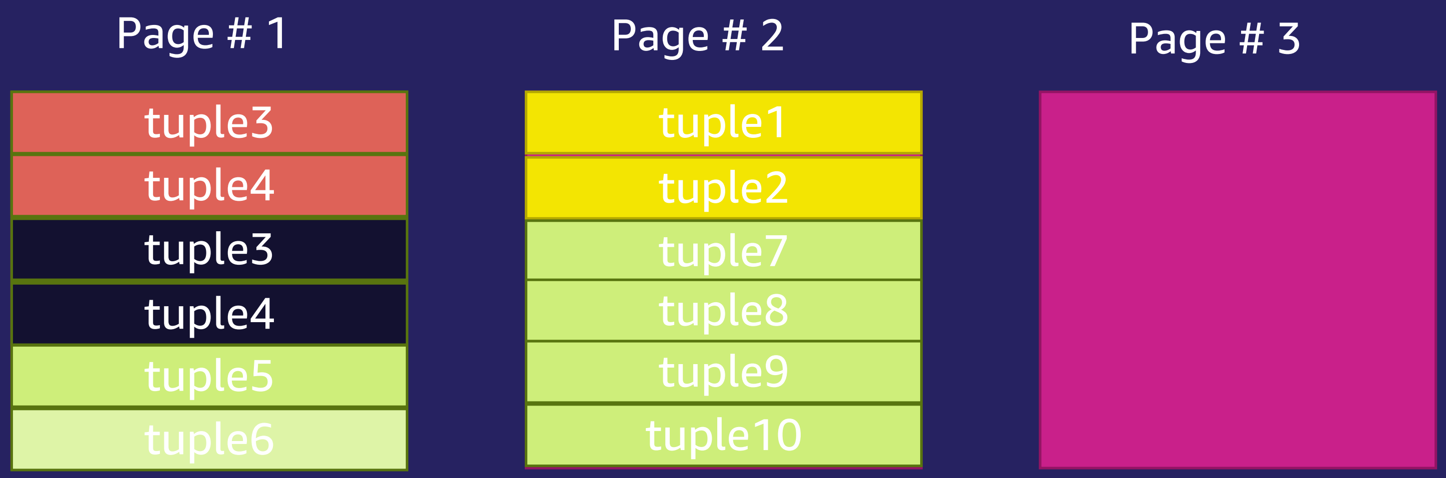
pg_hint_plan to bias queries away from big operations (hints related to Scans, Joins & Environment)

Autovacuum

In PostgreSQL, an UPDATE or DELETE operation doesn't immediately remove the old version of the row to gain benefits of Multi-Version Concurrency Control (MVCC)

AUTOVACUUM processes tables and related indexes on a regular basis

- To recover or reuse disk space occupied by updated or deleted rows.
    - Also defragments/rearranges rows on data pages to maintain contiguous free space
- To update data statistics used by the PostgreSQL query planner.
- To update the visibility map, which speeds up index-only scans.
- To protect against loss of very old data due to transaction ID wraparound or multixact ID wraparound.



Autovacuum runs

Autovacuum Issues

Indicator of Vacuuming issues

- Database storage growing with no new data influx?
- Noticing that your queries are running slow?
- Explain plan of a slow query shows sub-optimal plan (e.g. number of buffers read is much higher than number of actual rows returned) ?
- Maximum used transaction IDs constantly increasing beyond 200M transactions (by default)?

Detect

- List tables and its bloat ratio
- List indexes and its bloat ratio
- Implement Early Warning for Transaction ID Wraparound

Root causes

- Autovacuum not able to keep up
- Autovacuum getting blocked

Fixing Autovacuum Issues

- Adjust Autovacuum related parameters
  - Vacuuming related parameters can be set at the table level (using alter table <> set <>)
- Check and kill EXCLUSIVE locks on tables
- Check and kill "idle in transaction" session
- Check and kill long-running transactions
- Check and drop abandoned replication slots
- Check and rollback orphaned prepared transactions
- Run a manual vacuum (if needed)
  - Vacuum [table_name];
  - Vacuum ANALYZE [table_name];
  - Vacuum FULL [table_name];

Number of parameters to tune Autovacuum

```
vacuum_freeze_min_age
vacuum_freeze_table_age
autovacuum_freeze_max_age

autovacuum_max_workers
autovacuum_naptime
autovacuum_vacuum_cost_delay
autovacuum_vacuum_scale_factor
autovacuum_vacuum_cost_limit
maintenance_work_mem
```

https://aws.amazon.com/blogs/database/understanding-autovacuum-in-amazon-rds-for-postgresql-environments/

Optimizing Updates Using Fillfactor and HOT Updates

- Fillfactor specifies the % of a page to be filled by INSERT operations, reserving the rest of the space for subsequent UPDATE operations. Default Fillfactor for tables is 100% and for index is 90%.

- UPDATE operations insert a new row (or tuple) and mark the old row as dead.

- Every update by default requires new index entries to be added even if no indexed attribute is modified and modifying an index is much more expensive than modifying the table.

- Heavily updated tables can become “bloated” with dead tuples. Autovacuum operation cleans the dead row versions in the table and the index.

Optimizing Updates Using Fillfactor and HOT Updates

HOT (Heap Only Tuples) updates avoids updating index records by maintaining a chain of updated tuples linking a new version to the old in the data page.

Conditions

- New tuple is inserted in the same page as the old version of the tuple
- None of the indexed columns get changed

Advantages

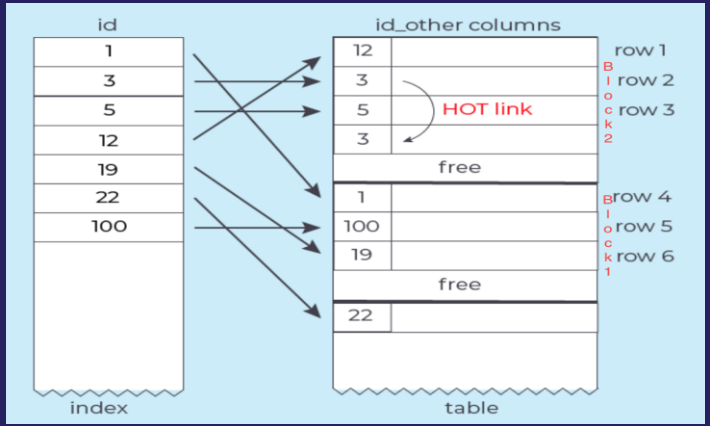
- UPDATEs are faster
- Dead tuples can be removed without the need for VACUUM. Any backend that processes a block and detects a HOT chain with dead tuples will try to lock and defragment the block, removing dead tuples.

Detect

Top30 tables with low HOT updates

Fix

1. ALTER TABLE <table_name> SET (fillfactor = 90)
2. Run pg_repack on the table to re-organize the table
3. Drop Unused, Duplicate and useless Indexes



HOT Updates

Optimizing Large Tables using Partitioning

Allows to split a large table into smaller pieces using List, Range or Hash partitioning techniques

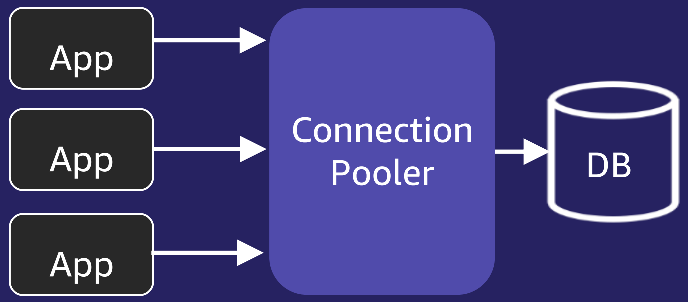
Benefits

- Partition Pruning: A query optimization technique where only a single partition or small number of partitions are accessed instead of all the partitions to fetch data to improve query performance
- Bulk loads and deletion can be done by adding or removing partitions which avoids Vacuum overhead
- Partition wise joins and partition wise aggregation
- Multiple vacuum workers can vacuum individual partitions in parallel

https://aws.amazon.com/blogs/database/improve-performance-and-manageability-of-large-postgresql-tables-by-migrating-to-partitioned-tables-on-amazon-aurora-and-amazon-rds/

Optimizing Connection overhead using Connection Pooling

- PostgreSQL has a postmaster process, which spawns new processes for each new connection to the database.
- Each open connection in PostgreSQL whether idle or active consumes memory (~10MB). This creates a problem if the number of connections are too high.
- Connection pooling refers to the method of creating a pool of connections and caching those connections for reuse.
- A database side connection pooler is recommended even if you have connection pool on the application side
- Connection poolers : RDS Proxy (fully managed and highly available), PgBouncer, Pgpool



PostgreSQL considerations for performance

Avoid using numeric datatype and consider bigint instead

- Numeric is designed for accurately storing monetary amounts. Can hold 131k digits before decimal and 16k digits after decimal.
- Joins and calculations on numeric columns are very slow compared to integer datatype.
- A simple pgbench test on numeric vs. bigint on write performance shows more than 15% difference.

Use limited number of Temporary tables

- Heavy usage can cause bloat in pg_catalog leading to slow performance and high CPU usage for queries touching dictionary tables.
- Monitor bloat in pg_catalog tables and tweak autovacuum to run aggressively if using temporary tables excessively.
- Autovacuum can't access temporary tables. So run Analyze on temporary tables after creation to help optimizer generate an optimal plan.

Pay attention to AUTOCOMMIT and "Idle in Transaction" session

- With autocommit OFF, even a select query opens a transaction and without implicit commit/rollback, transitions to idle in transaction state.
- "Idle in Transaction" session prevents autovacuum from cleaning up pages.
- Monitor and kill "Idle in Transaction" sessions or set idle_in_transaction_session_timeout parameter to kill these sessions automatically

Create separate Triggers for insert & update events and avoid using exception clauses

- Checking the value of TG_OP inside a trigger can be costly
- Each execution of an exception block results in allocation of an additional XID. This can rapidly exhaust transaction ids with high writes throughput.

Pay attention to the Volatility category (Volatile, Stable, and Immutable) of functions

- The Immutable variant takes the minimum amount of time.

Query Tuning in Aurora PostgreSQL

Query Tuning Methodology

1. Active Session Summary (Performance Insights, etc.)
2. Top SQL & Top Wait Events
3. EXPLAIN ANALYZE with Buffers, IO timing, etc.
4. Investigate STEP & WAIT taking the most time

Solving Problems with Wait Events

pg_stat_activity : One row per server process showing information related to the current activity of that process

pid	state	wait_event_type	wait_event	xact_runtime	query_short
8135	active			-00:00:00.000941	autovacuum: VACUUM pghist.pg_stat_statements_20190
8168	active			00:00:00	SELECT coll, col2,
108975		Activity	WalWriterMain		
108976		Activity	AutoVacuumMain		
108973		Activity	CheckpointerMain		
108974		Activity	BgWriterMain		
108979		Activity	LogicalLauncherMain		
8185	active			00:00:00.07941	autovacuum: VACUUM pghist.pg_stat_sys_indexes_2019
8212	active			00:00:00.349238	autovacuum: VACUUM pghist.pg_stat_statements_20190
115699	active	Lock	relation	00:30:01.170404	SELECT proc('param1')
103268	active	IO	DataFileRead	00:46:46.277548	select count(*) from some_ones_table a , (select c
95936	active	LWLock	buffer_io	00:56:57.327904	SELECT col1 FROM some_ones_table a, (SELECT col1 a
95935	active	IO	DataFileRead	00:56:57.328169	SELECT col1 FROM some_ones_table a, (SELECT col1 a
95921	active	LWLock	buffer_io	00:56:57.393765	SELECT col1 FROM some_ones_table a, (SELECT col1 a
56628	active	IO	DataFileRead	01:47:55.333596	select col1 from some_ones_table WHERE err_id in (
53981	active	IO	BufFileRead	01:51:40.986659	SELECT col1 FROM some_ones_table a, (SELECT asin a
49386	active	LWLock	buffer_io	01:58:13.166389	SELECT count(*) FROM some_ones_table a, (SELECT co
29172	active	IO	BufFileRead	02:04:09.108342	SELECT count(*) FROM some_ones_table a, (SELECT co
43208	active	LWLock	buffer_io	02:06:39.296499	SELECT count(*) FROM some_ones_table a, (SELECT co
43207	active	IO	DataFileRead	02:06:39.29666	SELECT count(*) FROM some_ones_table a, (SELECT co
31401	active	IPC	MessageQueueReceive	02:06:39.370239	SELECT count(*) FROM some_ones_table a, (SELECT co
12387	active	IO	DataFileRead	02:46:50.262871	select count(*) from some_ones_table a , (select c
12386	active	IO	DataFileRead	02:46:50.263142	select count(*) from some_ones_table a , (select c
12385	active	IO	DataFileRead	02:46:50.266696	select count(*) from some_ones_table a , (select c
83681	active	BufferPin	BufferPin	15:24:45.260184	autovacuum: VACUUM schema1.some_ones_table (to prev
23340	active	LWLock	buffer_io	1 day 16:39:18.732685	select column_001,column2,column3,column000004,
24074	active	LWLock	buffer_io	1 day 16:41:55.91496	WITH this_subquery_01 as (select column_001,PIPELI
8110	active	LWLock	buffer_io	1 day 17:03:52.767838	WITH this_subquery_01 as (select column_001,PIPEL
51767	active	LWLock	buffer_io	1 day 19:03:47.006302	WITH this_subquery_01 as (select column_001,PIPEL
9217	active	LWLock	buffer_io	1 day 20:01:58.572314	WITH this_subquery_01 as (select column_001,PIPEL
6086	active	IO	DataFileRead	1 day 20:06:08.584313	WITH this_subquery_01 as (select column_001,PIPEL
115385	active	LWLock	buffer_io	1 day 20:35:27.617606	WITH this_subquery_01 as (select column_001,PIPEL
94256	idle in trx	Client	ClientRead	27 days 02:33:48.940102	select subquery00_.column_001 as COLUMN01_2_0_, a

(33 rows)

aws

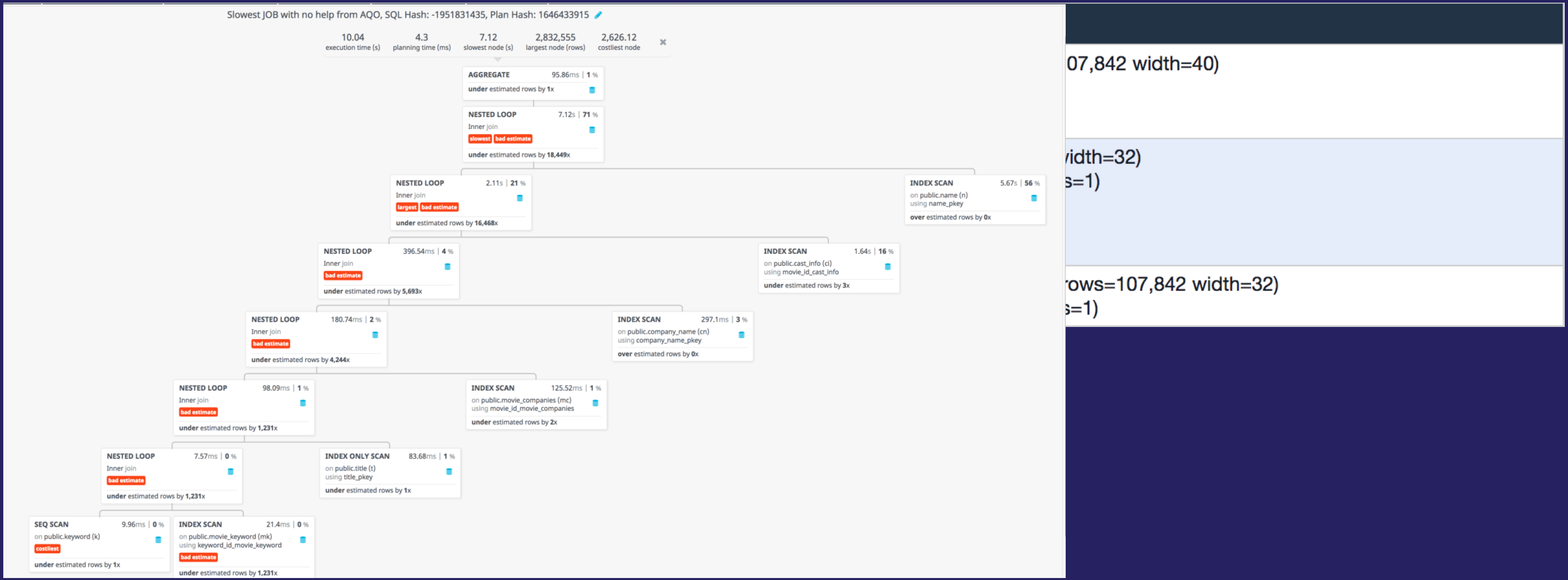
Explain Query Plan

explain (analyze,verbose,buffers,settings) <query>

⚠ Use transaction (begin, end) for running explain analyze on DML commands, so that you can rollback.

```
GroupAggregate  (cost=17612.84..19769.68 rows=107842 width=40) (actual time=861.091..884.817 rows=521 loops=1)
  Group Key: (st_geohash(geometry, 2))
  ->  Sort  (cost=17612.84..17882.44 rows=107842 width=32) (actual time=861.084..872.597 rows=107842 loops=1)
        Sort Key: (st_geohash(geometry, 2))
        Sort Method: external merge  Disk: 1376kB
        ->  Seq Scan on plan_item  (cost=0.00..6015.02 rows=107842 width=32) (actual time=0.018..50.245 rows=107842 loops=1)
Planning time: 0.094 ms
Execution time: 891.762 ms
```

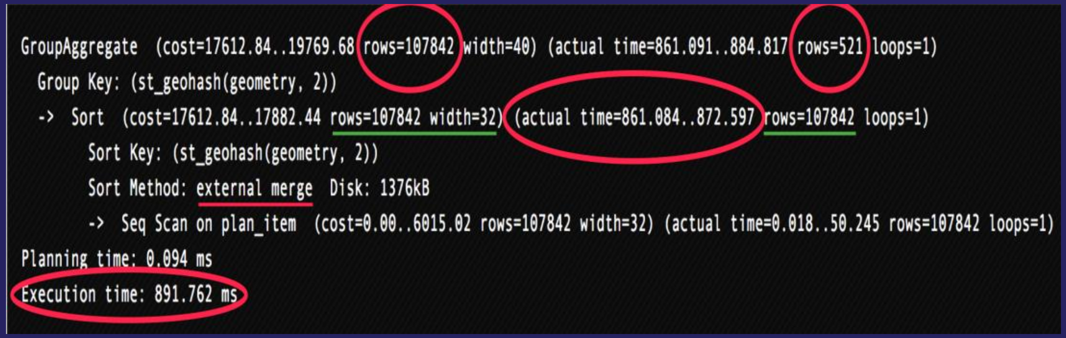
Visualize Query Plan



http://tatiyants.com/pev/#/plans/new

Problems to look for in EXPLAIN ANALYZE output

- Large difference between estimated and actual rows
- Wrong index, no index, or index not being used as expected
- Large number of buffers read (working set not cached)
- Slow nodes: Sort [Agg], NOT IN, OR, large SeqScan, COUNT
  - apg_enable_not_in_transform parameter in Aurora PostgreSQL
  - can help speed up NOT IN queries
- Bitmap heap scan reporting “lossy” (need to increase WORK_MEM)
- Large number of rows filtered by a post-join predicate
- Reading more data than necessary (pruning, clustering, index-only)
- Slow VOLATILE functions that are really IMMUTABLE



Query Plan Management (QPM)

1. Capture plans

   Automatically happens if query runs more than once

2. Approve plans

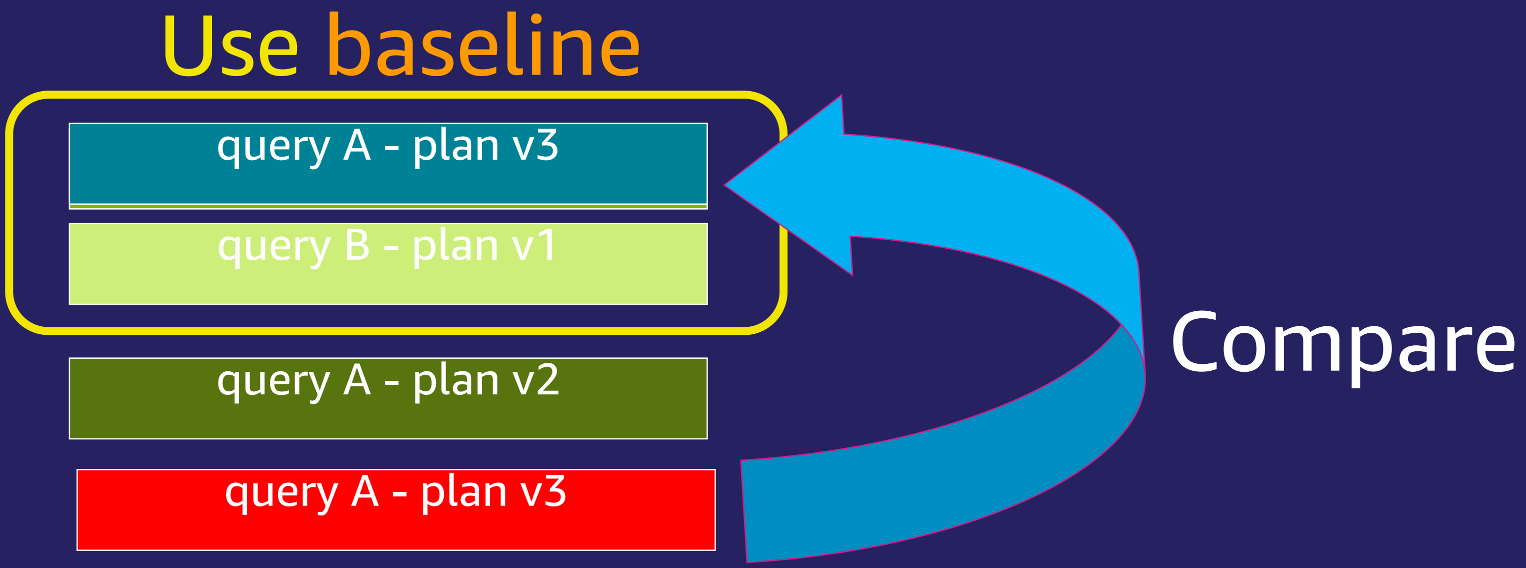
   First captured plan is automatically approved

3. Evolve Unapproved plans

   If an Unapproved plan is faster (slower), Approve (Reject) it.

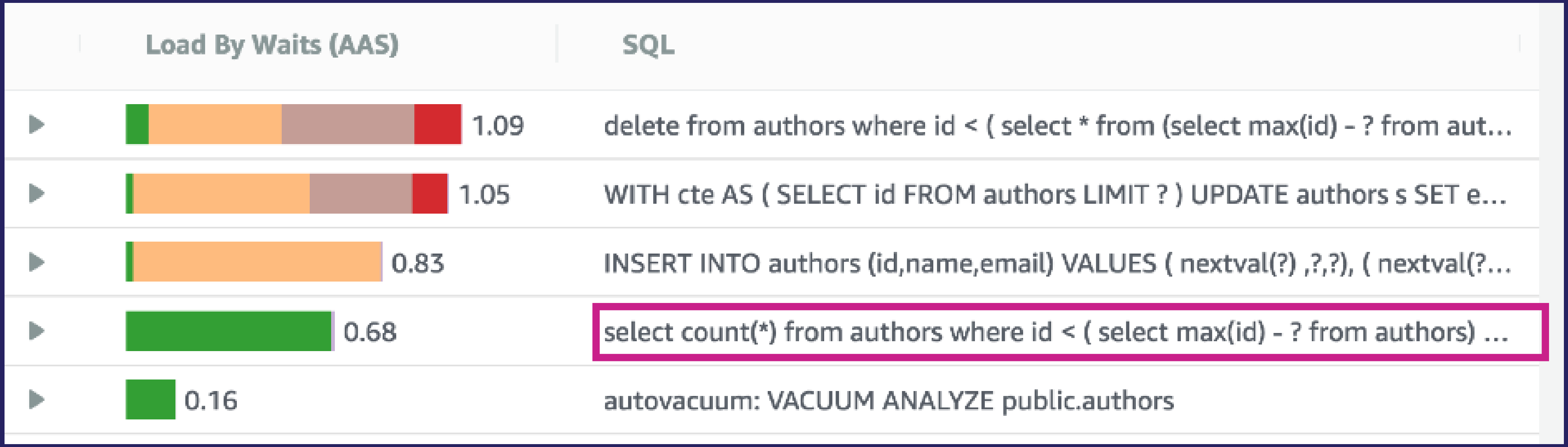
4. Re-test Approved plans and possibly change to Preferred or Rejected

5. See the effect of changing an optimizer setting for any set of statements, without risk of plan regression. Any new plans are created with status ‘Unapproved’.




```
SET work_mem = ‘4GB’;  -- try a different parameter setting
SELECT validate_plans (sql_hash, plan_hash, ‘’) FROM dba_plans
WHERE
        status in (‘Approved’, ‘Preferred’)  AND
        execution_time_ms >= 10000;
RESET work_mem;        -- restore the parameter to its default value
```

Analyzing a top statement from Performance Insights Using QPM

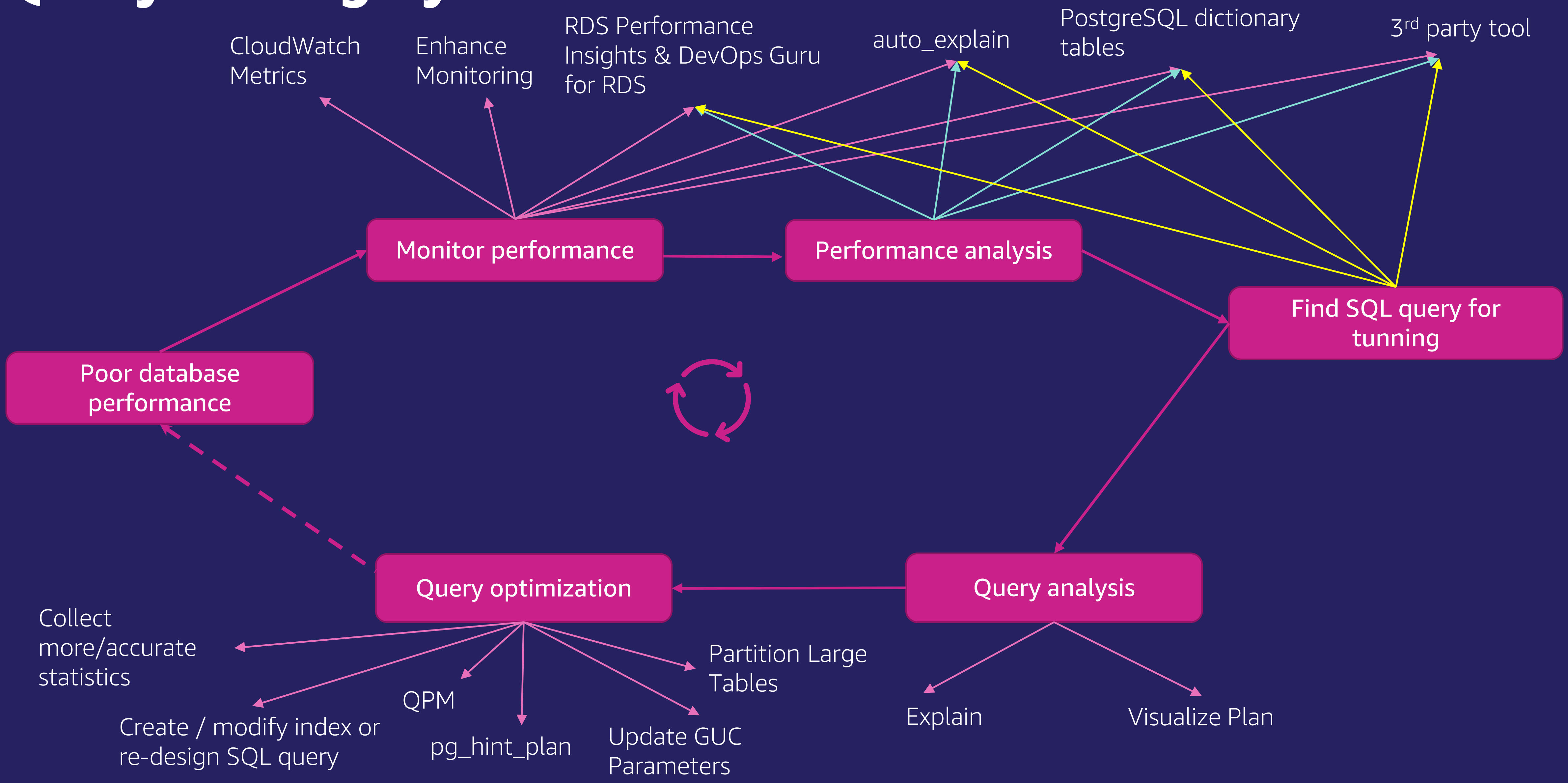



```
SELECT evolve_plan_baselines (sql_hash, plan_hash, 1.0, ‘approve’)
FROM dba_plans WHERE
  sql_text LIKE ‘select count(*) from authors where id < (select %’ AND
  plan_last_used (sql_hash, plan_hash) = current_date   -- used today
ORDER BY status DESC;   -- Unapproved first
```

Things you can do to make a slow query faster

- Collect more statistics (default_statistics_target) or extended statistics
- Modify parameters (GUCs) related to query planning and resource consumption (e.g. work_mem)
  - Review and Modify Aurora PostgreSQL specific optimizer parameters
- Fix the plan with pg_hint_plan, and *then remove the hint*
- Add secondary indexes, Foreign Key indexes and Drop unused indexes
  - Consider not only B-tree indexes, but also hash/BRIN/partial/expression indexes.
- Rewrite the SQL to a more efficiently executed form
- Reduce planning overhead or per-execution overhead (use prepared statements)
- CLUSTER cold parts of the heap to exploit access patterns
- Implement or change the table partitioning strategy
- Scale up to a larger instance class (to improve cache hit ratio)

Query Tuning Cycle



Partner Packages

Aurora Performance Optimization

Partner Packages – Aurora Performance Optimization

AGILISIUM
BIG ON CLOUD. BIG ON DATA.

Aurora Performance Optimization Offer

datAvail

Aurora Performance Optimization Offer

pwc

Aurora Performance Optimization Offer

virtusa®
Accelerating Business Outcomes

Aurora Performance Optimization Offer

Thank you!