Day 4 MySQL and PostgreSQL performance tuning

PostgreSql

postgresql is greet dbms in performance thanks to it MVCC that hold version of the rows to reduce conflict and deadlock
for today we have pagila already loaded and we will see how to to troubleshoot slow query
how to locate slow query and how to view query plan in more visual way

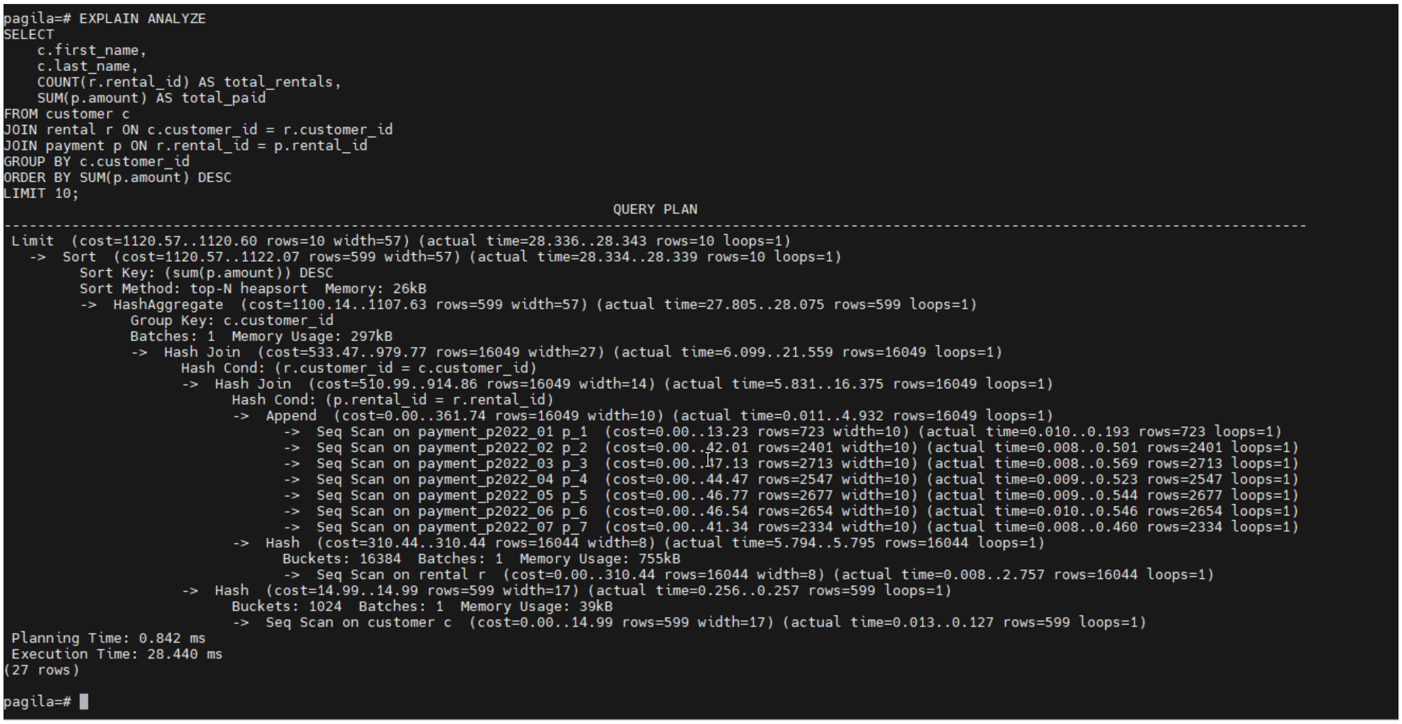
Query plan

we have the below query running on pagila database , you can execute once to confirm its working fine This query is intentionally written to be potentially inefficient

```
SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS total_rentals, SUM(p.amount) AS total_paid FROM customer c
JOIN rental r ON c.customer_id = r.customer_id JOIN payment p ON r.rental_id = p.rental_id GROUP BY c.customer_id,
c.first_name, c.last_name ORDER BY total_paid DESC LIMIT 10;
```

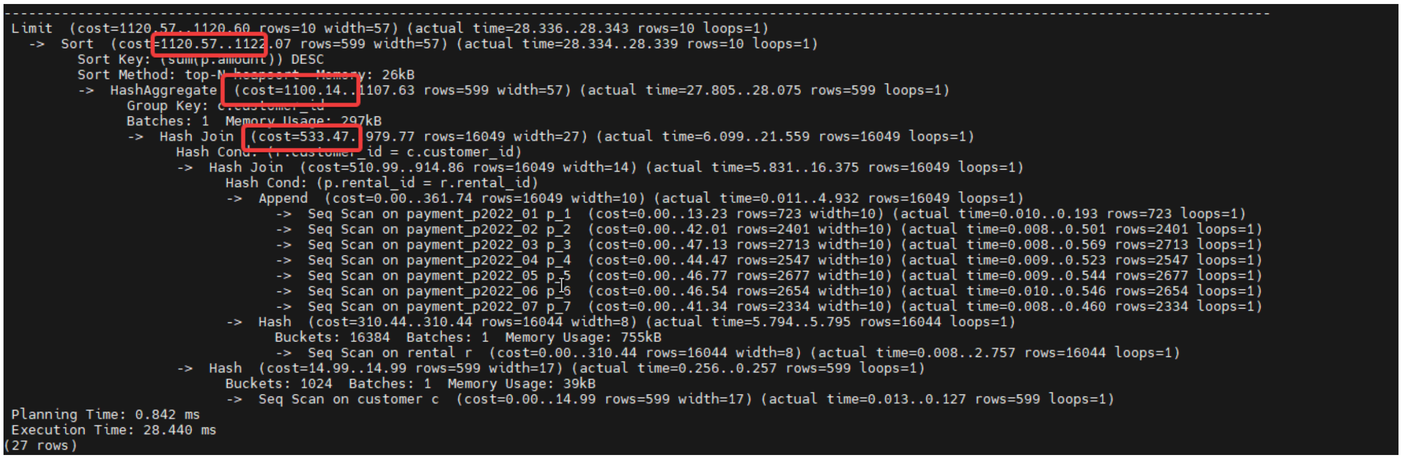
To Get the execution plan and actual runtime stats we will use `explain analyze` follow by the query it self

```
EXPLAIN ANALYZE SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS total_rentals, SUM(p.amount) AS total_paid
FROM customer c JOIN rental r ON c.customer_id = r.customer_id JOIN payment p ON r.rental_id = p.rental_id GROUP BY
c.customer_id ORDER BY SUM(p.amount) DESC LIMIT 10;
```



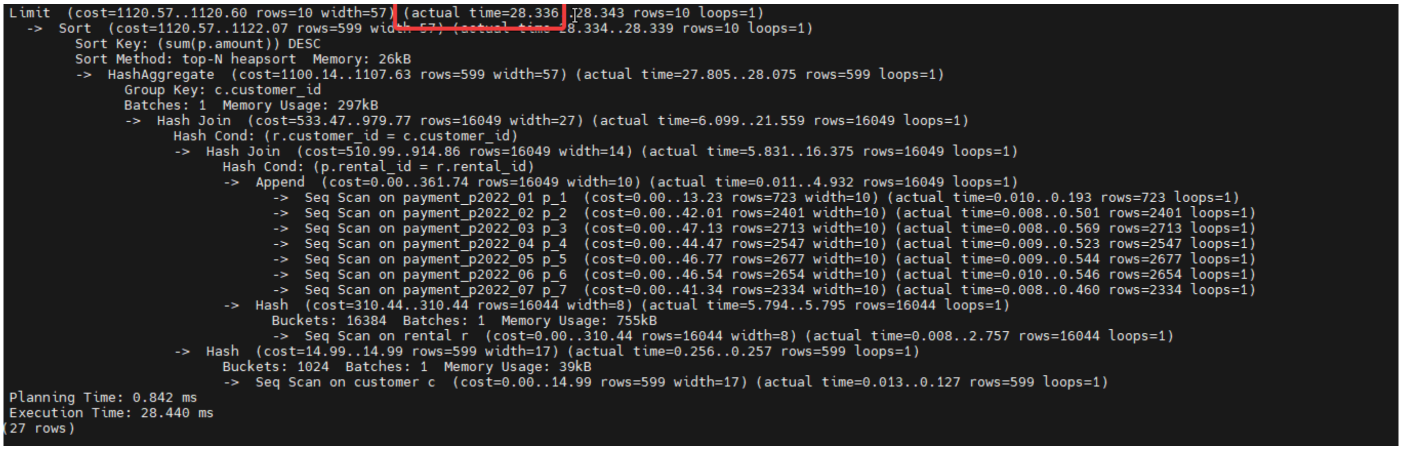
```
pagila=# EXPLAIN ANALYZE
SELECT
    c.first_name,
    c.last_name,
    COUNT(r.rental_id) AS total_rentals,
    SUM(p.amount) AS total_paid
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
JOIN payment p ON r.rental_id = p.rental_id
GROUP BY c.customer_id
ORDER BY SUM(p.amount) DESC
LIMIT 10;
                                                                           QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=1120.57..1120.60 rows=10 width=57) (actual time=28.336..28.343 rows=10 loops=1)
   ->  Sort  (cost=1120.57..1122.07 rows=599 width=57) (actual time=28.334..28.339 rows=10 loops=1)
         Sort Key: (sum(p.amount)) DESC
         Sort Method: top-N heapsort  Memory: 26kB
         ->  HashAggregate  (cost=1100.14..1107.63 rows=599 width=57) (actual time=27.805..28.075 rows=599 loops=1)
               Group Key: c.customer_id
               Batches: 1  Memory Usage: 297kB
               ->  Hash Join  (cost=533.47..979.77 rows=16049 width=27) (actual time=6.099..21.559 rows=16049 loops=1)
                     Hash Cond: (r.customer_id = c.customer_id)
                     ->  Hash Join  (cost=510.99..914.86 rows=16049 width=14) (actual time=5.831..16.375 rows=16049 loops=1)
                           Hash Cond: (p.rental_id = r.rental_id)
                           ->  Append  (cost=0.00..361.74 rows=16049 width=10) (actual time=0.011..4.932 rows=16049 loops=1)
                                 ->  Seq Scan on payment_p2022_01 p_1  (cost=0.00..13.23 rows=723 width=10) (actual time=0.010..0.193 rows=723 loops=1)
                                 ->  Seq Scan on payment_p2022_02 p_2  (cost=0.00..42.01 rows=2401 width=10) (actual time=0.008..0.501 rows=2401 loops=1)
                                 ->  Seq Scan on payment_p2022_03 p_3  (cost=0.00..47.13 rows=2713 width=10) (actual time=0.008..0.569 rows=2713 loops=1)
                                 ->  Seq Scan on payment_p2022_04 p_4  (cost=0.00..44.47 rows=2547 width=10) (actual time=0.009..0.523 rows=2547 loops=1)
                                 ->  Seq Scan on payment_p2022_05 p_5  (cost=0.00..46.77 rows=2677 width=10) (actual time=0.009..0.544 rows=2677 loops=1)
                                 ->  Seq Scan on payment_p2022_06 p_6  (cost=0.00..46.54 rows=2654 width=10) (actual time=0.010..0.546 rows=2654 loops=1)
                                 ->  Seq Scan on payment_p2022_07 p_7  (cost=0.00..41.34 rows=2334 width=10) (actual time=0.008..0.460 rows=2334 loops=1)
                           ->  Hash  (cost=310.44..310.44 rows=16044 width=8) (actual time=5.794..5.795 rows=16044 loops=1)
                                 Buckets: 16384  Batches: 1  Memory Usage: 755kB
                                 ->  Seq Scan on rental r  (cost=0.00..310.44 rows=16044 width=8) (actual time=0.008..2.757 rows=16044 loops=1)
                     ->  Hash  (cost=14.99..14.99 rows=599 width=17) (actual time=0.256..0.257 rows=599 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 39kB
                           ->  Seq Scan on customer c  (cost=0.00..14.99 rows=599 width=17) (actual time=0.013..0.127 rows=599 loops=1)
 Planning Time: 0.842 ms
 Execution Time: 28.440 ms
(27 rows)

pagila=#
```

in analyze output look for high cost value



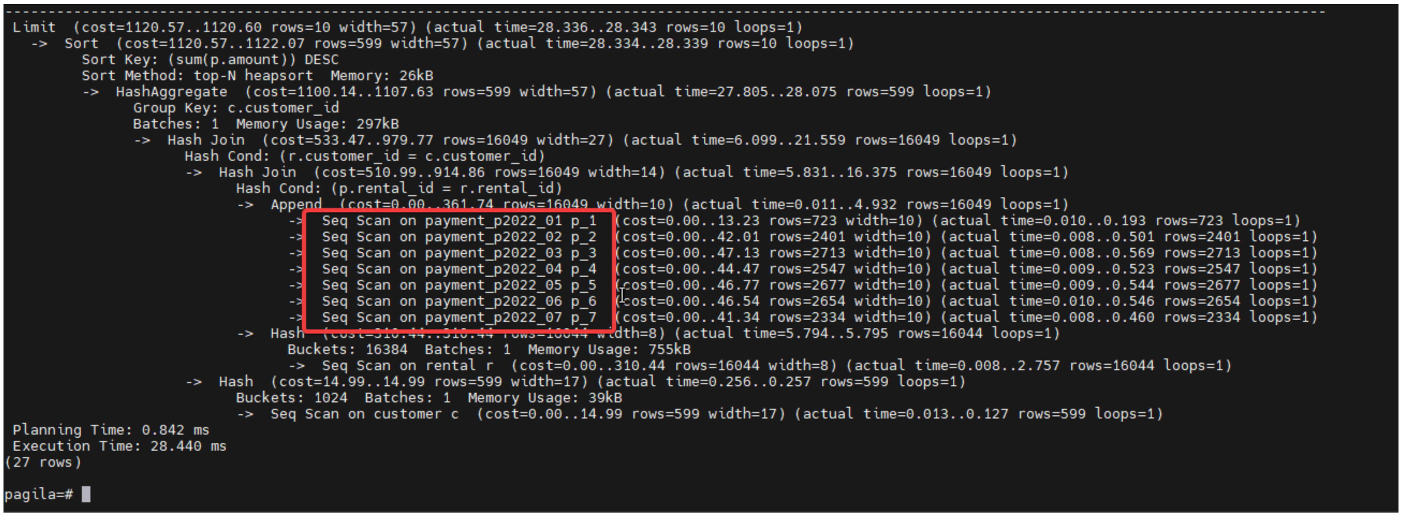
```
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=1120.57..1120.60 rows=10 width=57) (actual time=28.336..28.343 rows=10 loops=1)
   ->  Sort  (cost=1120.57..1122.07 rows=599 width=57) (actual time=28.334..28.339 rows=10 loops=1)
         Sort Key: (sum(p.amount)) DESC
         Sort Method: top-N heapsort  Memory: 26kB
         ->  HashAggregate  (cost=1100.14..1107.63 rows=599 width=57) (actual time=27.805..28.075 rows=599 loops=1)
               Group Key: c.customer_id
               Batches: 1  Memory Usage: 297kB
               ->  Hash Join  (cost=533.47..979.77 rows=16049 width=27) (actual time=6.099..21.559 rows=16049 loops=1)
                     Hash Cond: (r.customer_id = c.customer_id)
                     ->  Hash Join  (cost=510.99..914.86 rows=16049 width=14) (actual time=5.831..16.375 rows=16049 loops=1)
                           Hash Cond: (p.rental_id = r.rental_id)
                           ->  Append  (cost=0.00..361.74 rows=16049 width=10) (actual time=0.011..4.932 rows=16049 loops=1)
                                 ->  Seq Scan on payment_p2022_01 p_1  (cost=0.00..13.23 rows=723 width=10) (actual time=0.010..0.193 rows=723 loops=1)
                                 ->  Seq Scan on payment_p2022_02 p_2  (cost=0.00..42.01 rows=2401 width=10) (actual time=0.008..0.501 rows=2401 loops=1)
                                 ->  Seq Scan on payment_p2022_03 p_3  (cost=0.00..47.13 rows=2713 width=10) (actual time=0.008..0.569 rows=2713 loops=1)
                                 ->  Seq Scan on payment_p2022_04 p_4  (cost=0.00..44.47 rows=2547 width=10) (actual time=0.009..0.523 rows=2547 loops=1)
                                 ->  Seq Scan on payment_p2022_05 p_5  (cost=0.00..46.77 rows=2677 width=10) (actual time=0.009..0.544 rows=2677 loops=1)
                                 ->  Seq Scan on payment_p2022_06 p_6  (cost=0.00..46.54 rows=2654 width=10) (actual time=0.010..0.546 rows=2654 loops=1)
                                 ->  Seq Scan on payment_p2022_07 p_7  (cost=0.00..41.34 rows=2334 width=10) (actual time=0.008..0.460 rows=2334 loops=1)
                           ->  Hash  (cost=310.44..310.44 rows=16044 width=8) (actual time=5.794..5.795 rows=16044 loops=1)
                                 Buckets: 16384  Batches: 1  Memory Usage: 755kB
                                 ->  Seq Scan on rental r  (cost=0.00..310.44 rows=16044 width=8) (actual time=0.008..2.757 rows=16044 loops=1)
                     ->  Hash  (cost=14.99..14.99 rows=599 width=17) (actual time=0.256..0.257 rows=599 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 39kB
                           ->  Seq Scan on customer c  (cost=0.00..14.99 rows=599 width=17) (actual time=0.013..0.127 rows=599 loops=1)
 Planning Time: 0.842 ms
 Execution Time: 28.440 ms
(27 rows)
```

look for high actual time The time in milliseconds spent on that step



```
 Limit  (cost=1120.57..1120.60 rows=10 width=57) (actual time=28.336..28.343 rows=10 loops=1)
   ->  Sort  (cost=1120.57..1122.07 rows=599 width=57) (actual time=28.334..28.339 rows=10 loops=1)
         Sort Key: (sum(p.amount)) DESC
         Sort Method: top-N heapsort  Memory: 26kB
         ->  HashAggregate  (cost=1100.14..1107.63 rows=599 width=57) (actual time=27.805..28.075 rows=599 loops=1)
               Group Key: c.customer_id
               Batches: 1  Memory Usage: 297kB
               ->  Hash Join  (cost=533.47..979.77 rows=16049 width=27) (actual time=6.099..21.559 rows=16049 loops=1)
                     Hash Cond: (r.customer_id = c.customer_id)
                     ->  Hash Join  (cost=510.99..914.86 rows=16049 width=14) (actual time=5.831..16.375 rows=16049 loops=1)
                           Hash Cond: (p.rental_id = r.rental_id)
                           ->  Append  (cost=0.00..361.74 rows=16049 width=10) (actual time=0.011..4.932 rows=16049 loops=1)
                                 ->  Seq Scan on payment_p2022_01 p_1  (cost=0.00..13.23 rows=723 width=10) (actual time=0.010..0.193 rows=723 loops=1)
                                 ->  Seq Scan on payment_p2022_02 p_2  (cost=0.00..42.01 rows=2401 width=10) (actual time=0.008..0.501 rows=2401 loops=1)
                                 ->  Seq Scan on payment_p2022_03 p_3  (cost=0.00..47.13 rows=2713 width=10) (actual time=0.008..0.569 rows=2713 loops=1)
                                 ->  Seq Scan on payment_p2022_04 p_4  (cost=0.00..44.47 rows=2547 width=10) (actual time=0.009..0.523 rows=2547 loops=1)
                                 ->  Seq Scan on payment_p2022_05 p_5  (cost=0.00..46.77 rows=2677 width=10) (actual time=0.009..0.544 rows=2677 loops=1)
                                 ->  Seq Scan on payment_p2022_06 p_6  (cost=0.00..46.54 rows=2654 width=10) (actual time=0.010..0.546 rows=2654 loops=1)
                                 ->  Seq Scan on payment_p2022_07 p_7  (cost=0.00..41.34 rows=2334 width=10) (actual time=0.008..0.460 rows=2334 loops=1)
                           ->  Hash  (cost=310.44..310.44 rows=16044 width=8) (actual time=5.794..5.795 rows=16044 loops=1)
                                 Buckets: 16384  Batches: 1  Memory Usage: 755kB
                                 ->  Seq Scan on rental r  (cost=0.00..310.44 rows=16044 width=8) (actual time=0.008..2.757 rows=16044 loops=1)
                     ->  Hash  (cost=14.99..14.99 rows=599 width=17) (actual time=0.256..0.257 rows=599 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 39kB
                           ->  Seq Scan on customer c  (cost=0.00..14.99 rows=599 width=17) (actual time=0.013..0.127 rows=599 loops=1)
 Planning Time: 0.842 ms
 Execution Time: 28.440 ms
(27 rows)
```

look for A Seq Scan (Sequential Scan) means the table is getting fully scanned might require to create index on the column you can see from pic the scan is done on payment column



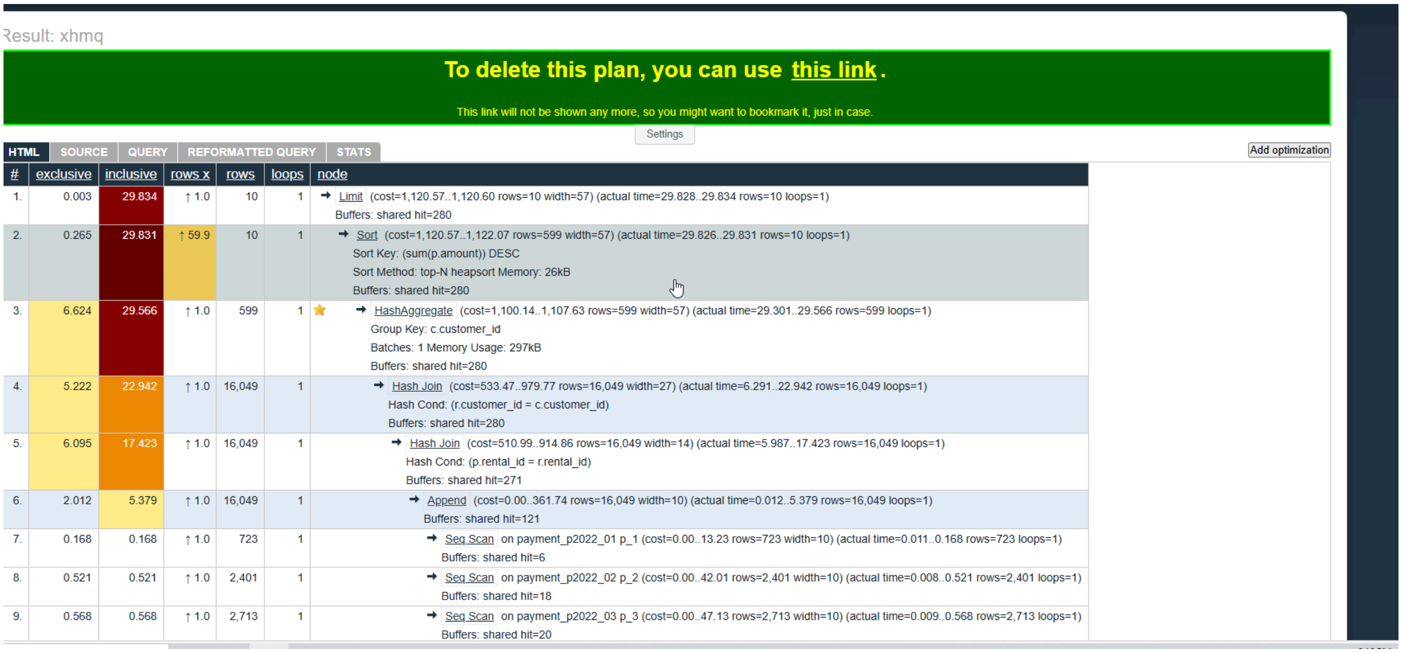
```
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=1120.57..1120.60 rows=10 width=57) (actual time=28.336..28.343 rows=10 loops=1)
   ->  Sort  (cost=1120.57..1122.07 rows=599 width=57) (actual time=28.334..28.339 rows=10 loops=1)
         Sort Key: (sum(p.amount)) DESC
         Sort Method: top-N heapsort  Memory: 26kB
         ->  HashAggregate  (cost=1100.14..1107.63 rows=599 width=57) (actual time=27.805..28.075 rows=599 loops=1)
               Group Key: c.customer_id
               Batches: 1  Memory Usage: 297kB
               ->  Hash Join  (cost=533.47..979.77 rows=16049 width=27) (actual time=6.099..21.559 rows=16049 loops=1)
                     Hash Cond: (r.customer_id = c.customer_id)
                     ->  Hash Join  (cost=510.99..914.86 rows=16049 width=14) (actual time=5.831..16.375 rows=16049 loops=1)
                           Hash Cond: (p.rental_id = r.rental_id)
                           ->  Append  (cost=0.00..361.74 rows=16049 width=10) (actual time=0.011..4.932 rows=16049 loops=1)
                                 ->  Seq Scan on payment_p2022_01 p_1  (cost=0.00..13.23 rows=723 width=10) (actual time=0.010..0.193 rows=723 loops=1)
                                 ->  Seq Scan on payment_p2022_02 p_2  (cost=0.00..42.01 rows=2401 width=10) (actual time=0.008..0.501 rows=2401 loops=1)
                                 ->  Seq Scan on payment_p2022_03 p_3  (cost=0.00..47.13 rows=2713 width=10) (actual time=0.008..0.569 rows=2713 loops=1)
                                 ->  Seq Scan on payment_p2022_04 p_4  (cost=0.00..44.47 rows=2547 width=10) (actual time=0.009..0.523 rows=2547 loops=1)
                                 ->  Seq Scan on payment_p2022_05 p_5  (cost=0.00..46.77 rows=2677 width=10) (actual time=0.009..0.544 rows=2677 loops=1)
                                 ->  Seq Scan on payment_p2022_06 p_6  (cost=0.00..46.54 rows=2654 width=10) (actual time=0.010..0.546 rows=2654 loops=1)
                                 ->  Seq Scan on payment_p2022_07 p_7  (cost=0.00..41.34 rows=2334 width=10) (actual time=0.008..0.460 rows=2334 loops=1)
                           ->  Hash  (cost=310.44..310.44 rows=16044 width=8) (actual time=5.794..5.795 rows=16044 loops=1)
                                 Buckets: 16384  Batches: 1  Memory Usage: 755kB
                                 ->  Seq Scan on rental r  (cost=0.00..310.44 rows=16044 width=8) (actual time=0.008..2.757 rows=16044 loops=1)
                     ->  Hash  (cost=14.99..14.99 rows=599 width=17) (actual time=0.256..0.257 rows=599 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 39kB
                           ->  Seq Scan on customer c  (cost=0.00..14.99 rows=599 width=17) (actual time=0.013..0.127 rows=599 loops=1)
 Planning Time: 0.842 ms
 Execution Time: 28.440 ms
(27 rows)

pagila=# 
```

Visualizing the Query Plan

show query plan in text output is not greet way to analyze the performance and for huge query it will take more time from your side to analyze this why its best to visualize the query plan , one greet website is https://explain.depesz.com/ which allow you to past explain analyze output and then it will visualize the query
, run `EXPLAIN (ANALYZE, BUFFERS)` follow by the query and past the output to the website

```
EXPLAIN (ANALYZE, BUFFERS) SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS total_rentals, SUM(p.amount) AS
total_paid FROM customer c JOIN rental r ON c.customer_id = r.customer_id JOIN payment p ON r.rental_id = p.rental_id
```

```
GROUP BY c.customer_id ORDER BY SUM(p.amount) DESC LIMIT 10;
```



Result: xhmq
To delete this plan, you can use this link.
This link will not be shown any more, so you might want to bookmark it, just in case.
Settings
HTML
SOURCE
QUERY
REFORMATTED QUERY
STATS
Add optimization

#	exclusive	inclusive	rows x	rows	loops	node
1.	0.003	29.834	↑ 1.0	10	1	→ Limit (cost=1,120.57..1,120.60 rows=10 width=57) (actual time=29.828..29.834 rows=10 loops=1) Buffers: shared hit=280
2.	0.265	29.831	↑ 59.9	10	1	→ Sort (cost=1,120.57..1,122.07 rows=599 width=57) (actual time=29.826..29.831 rows=10 loops=1) Sort Key: (sum(p.amount)) DESC Sort Method: top-N heapsort Memory: 26kB Buffers: shared hit=280
3.	6.624	29.566	↑ 1.0	599	1	★ → HashAggregate (cost=1,100.14..1,107.63 rows=599 width=57) (actual time=29.301..29.566 rows=599 loops=1) Group Key: c.customer_id Batches: 1 Memory Usage: 297kB Buffers: shared hit=280
4.	5.222	22.942	↑ 1.0	16,049	1	→ Hash Join (cost=533.47..979.77 rows=16,049 width=27) (actual time=6.291..22.942 rows=16,049 loops=1) Hash Cond: (r.customer_id = c.customer_id) Buffers: shared hit=280
5.	6.095	17.423	↑ 1.0	16,049	1	→ Hash Join (cost=510.99..914.86 rows=16,049 width=14) (actual time=5.987..17.423 rows=16,049 loops=1) Hash Cond: (p.rental_id = r.rental_id) Buffers: shared hit=271
6.	2.012	5.379	↑ 1.0	16,049	1	→ Append (cost=0.00..361.74 rows=16,049 width=10) (actual time=0.012..5.379 rows=16,049 loops=1) Buffers: shared hit=121
7.	0.168	0.168	↑ 1.0	723	1	→ Seq Scan on payment_p2022_01 p_1 (cost=0.00..13.23 rows=723 width=10) (actual time=0.011..0.168 rows=723 loops=1) Buffers: shared hit=6
8.	0.521	0.521	↑ 1.0	2,401	1	→ Seq Scan on payment_p2022_02 p_2 (cost=0.00..42.01 rows=2,401 width=10) (actual time=0.008..0.521 rows=2,401 loops=1) Buffers: shared hit=18
9.	0.568	0.568	↑ 1.0	2,713	1	→ Seq Scan on payment_p2022_03 p_3 (cost=0.00..47.13 rows=2,713 width=10) (actual time=0.009..0.568 rows=2,713 loops=1) Buffers: shared hit=20

you can see the it will mark problems in query plan and from there you can understand which section from query is taking more time to execute
for this query `limit` and `sort` are taking mush of the query time so it need to be optimized
you might need to check column that is sorting or limit or hashing and create index on column

note that `explain analyze` will execute the query to provide live state from query plan
if you don't want to execute query you can use `explain` follow by query it will return estimated executing plan

identify slow query

to identify poor performance query you need it add `pg_stat_statment` in preloaded library in `postgresql.conf` , if you follow day 1 installing and configuring PostgreSQL you saw that we have edited PostgreSQL parameter and enabled also pg_state_statment

best start point is to run the below query it will show what session running query what their username and client address and wait event causing delay for query executing

```
SELECT pid, usename, client_addr, ,client_addr,state,  wait_event  FROM  pg_stat_activity WHERE state = 'active' AND
pid <> pg_backend_pid();
```

below query also useful to see if query is wating for quirng lock and
it will; show blocked pid and blocking pid

```
 SELECT
    -- Details of the process holding the lock
    pl.pid AS blocking_pid,
    pa.query AS blocking_query,

    -- Details of the process being blocked
    al.pid AS blocked_pid,
    a.query AS blocked_query,
    to_char(a.query_start, 'YYYY-MM-DD HH24:MI:SS') as
blocked_query_started
FROM
    pg_locks al
JOIN
    pg_stat_activity a ON al.pid = a.pid
JOIN
```

```
    pg_locks pl ON al.locktype = pl.locktype AND al.database is not
distinct from pl.database AND al.relation is not distinct from
pl.relation AND al.page is not distinct from pl.page AND al.tuple is
not distinct from pl.tuple AND al.virtualxid is not distinct from
pl.virtualxid AND al.transactionid is not distinct from
pl.transactionid AND al.classid is not distinct from pl.classid AND
al.objid is not distinct from pl.objid AND al.objsubid is not distinct
from pl.objsubid AND al.pid <> pl.pid
JOIN
    pg_stat_activity pa ON pl.pid = pa.pid
WHERE
    NOT al.granted;
```

What is `pg_stat_statements`?

It is a PostgreSQL extension that tracks execution statistics for all SQL statements executed on your server. For every unique query, it records key metrics like:

- How many times the query was executed.
- The total time spent executing that query.
- The average execution time.
- How much data it read from memory vs. disk.
- How many rows it returned.

This information is invaluable for identifying slow, resource-intensive, or frequently run queries that are prime candidates for optimization.

to start we need to enable extension on database

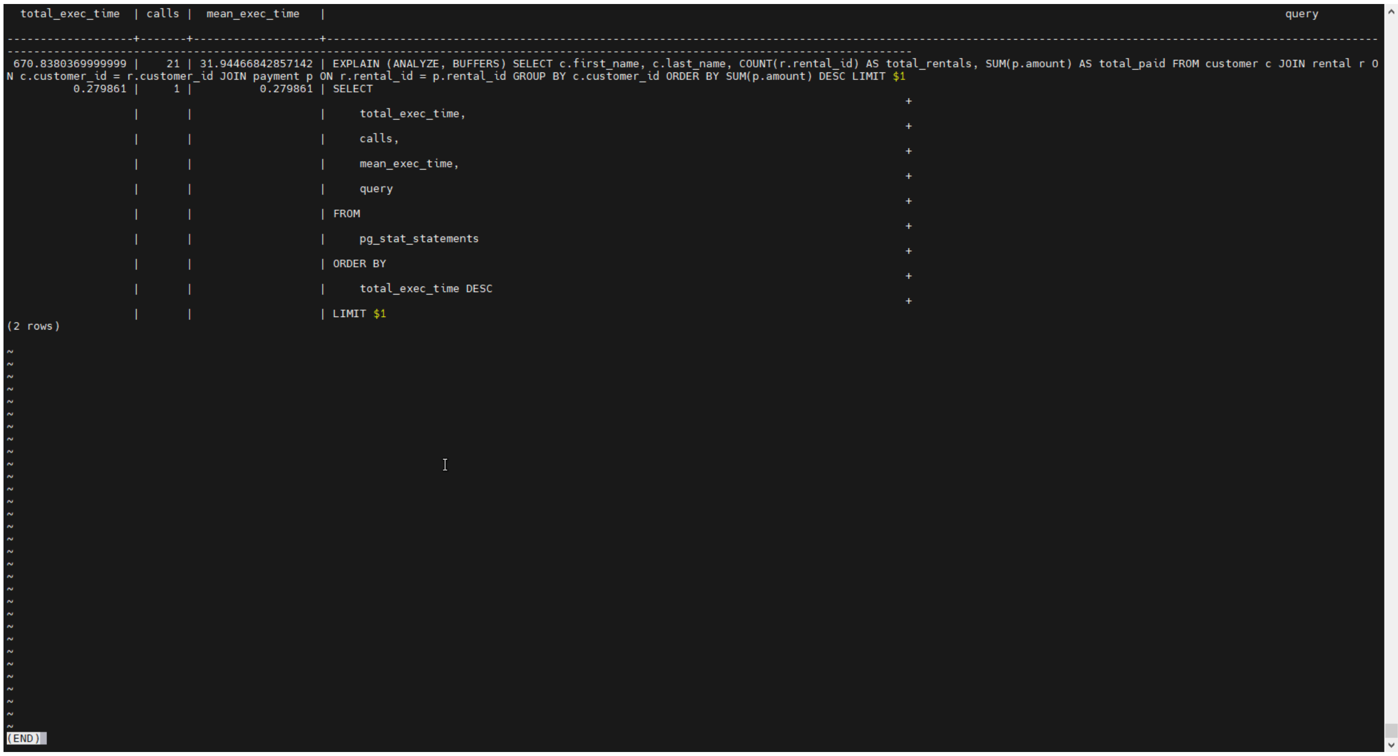
```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
```

then we can run varius query to retrive query base on metric we specify

Query 1: Find the Top 10 Most Time-Consuming Queries Overall

This query shows which queries are responsible for the most cumulative time spent on the server. These are often the best candidates for optimization.

```
SELECT
    total_exec_time,
    calls,
    mean_exec_time,
    query
FROM
    pg_stat_statements
ORDER BY
    total_exec_time DESC
LIMIT 10;
```



```
  total_exec_time  | calls |  mean_exec_time   |                                                                                                                                                           query

-------------------+-------+-------------------+-------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 670.8380369999999 |    21 | 31.94466842857142 | EXPLAIN (ANALYZE, BUFFERS) SELECT c.first_name, c.last_name, COUNT(r.rental_id) AS total_rentals, SUM(p.amount) AS total_paid FROM customer c JOIN rental r O
N c.customer_id = r.customer_id JOIN payment p ON r.rental_id = p.rental_id GROUP BY c.customer_id ORDER BY SUM(p.amount) DESC LIMIT $1
          0.279861 |     1 |          0.279861 | SELECT
                                                                                                                                       +
                   |       |                   |     total_exec_time,
                                                                                                                                       +
                   |       |                   |     calls,
                                                                                                                                       +
                   |       |                   |     mean_exec_time,
                                                                                                                                       +
                   |       |                   |     query
                                                                                                                                       +
                   |       |                   | FROM
                                                                                                                                       +
                   |       |                   |     pg_stat_statements
                                                                                                                                       +
                   |       |                   | ORDER BY
                                                                                                                                       +
                   |       |                   |     total_exec_time DESC
                                                                                                                                       +
                   |       |                   | LIMIT $1
(2 rows)

~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
(END)
```

Query 2: Find the Top 10 Most Frequently Executed Queries

This helps you find "chatty" application behavior. A query might be very fast individually, but if it's called millions of times, it can create significant load.

```
SELECT
    calls,
    total_exec_time,
    mean_exec_time,
    query
FROM
    pg_stat_statements
ORDER BY
    calls DESC
LIMIT 10;
```

Query 2: Find the Top 10 Most Frequently Executed Queries

This helps you find "chatty" application behavior. A query might be very fast individually, but if it's called millions of times, it can create significant load.

SQL

```
SELECT
    calls,
    total_exec_time,
    mean_exec_time,
    query
FROM
    pg_stat_statements
ORDER BY
    calls DESC
LIMIT 10;
```

Query 3: Find Queries that Read the Most from Disk

Queries that read a lot from disk (instead of from memory/cache) are often slow due to I/O waits. This can indicate missing indexes.

SQL

```
SELECT
    (shared_blks_read + local_blks_read + temp_blks_read) AS total_disk_reads,
    query
FROM
    pg_stat_statements
ORDER BY
    total_disk_reads DESC
LIMIT 10;
```

- `shared_blks_read` : Data read from disk for your tables/indexes. **High numbers here often point to missing indexes.**
- `temp_blks_read` : Data read from temporary on-disk files, often caused by large sorts or joins that don't fit in `work_mem` .
  Resetting the Statistics

After you perform a major optimization (like adding an index or rewriting a query), you may want to reset the statistics to get a fresh baseline.

You can clear all collected statistics by running:

SQL

```
SELECT pg_stat_statements_reset();
```

MySQL

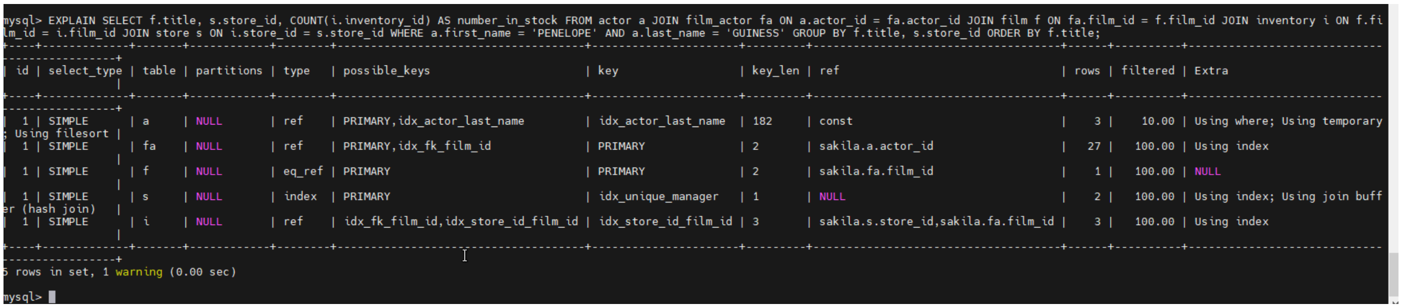
for MySQL its have system views called performance schema that hold statics for everything running in MySQL
in addition showing the executing plan for query is very similar to PostgreSQL
you use `explain analyze` follow by query for live state
or `explain` follow by the query for estimated query plan

This query filters by an actor's first and last name, which is often a source of inefficiency if not indexed correctly.
-- Find all films for a specific actor and their stock levels

```
SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON a.actor_id =
fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.film_id = i.film_id JOIN store s ON
i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id
ORDER BY f.title;
```

Get the execution plan:

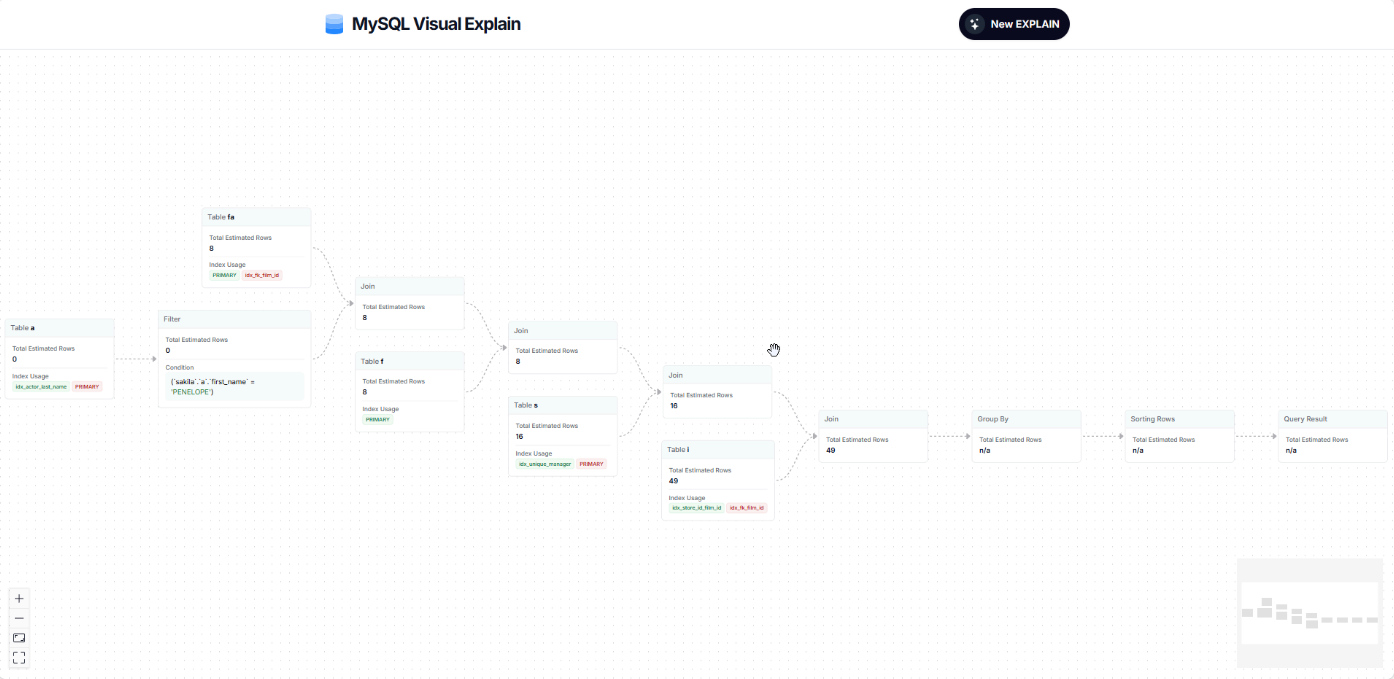
```

EXPLAIN SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON
a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.film_id = i.film_id JOIN store s
ON i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id
ORDER BY f.title;
```



```
mysql> EXPLAIN SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.fi
lm_id = i.film_id JOIN store s ON i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id ORDER BY f.title;
+----+-------------+-------+------------+--------+--------------------------------------+----------------------+---------+-------------------------------------+------+----------+-----------------------------
-----------------+
| id | select_type | table | partitions | type   | possible_keys                        | key                  | key_len | ref                                 | rows | filtered | Extra
                 |
+----+-------------+-------+------------+--------+--------------------------------------+----------------------+---------+-------------------------------------+------+----------+-----------------------------
-----------------+
|  1 | SIMPLE      | a     | NULL       | ref    | PRIMARY,idx_actor_last_name          | idx_actor_last_name  | 182     | const                               |    3 |    10.00 | Using where; Using temporary
; Using filesort |
|  1 | SIMPLE      | fa    | NULL       | ref    | PRIMARY,idx_fk_film_id               | PRIMARY              | 2       | sakila.a.actor_id                   |   27 |   100.00 | Using index
                 |
|  1 | SIMPLE      | f     | NULL       | eq_ref | PRIMARY                              | PRIMARY              | 2       | sakila.fa.film_id                   |    1 |   100.00 | NULL
                 |
|  1 | SIMPLE      | s     | NULL       | index  | PRIMARY                              | idx_unique_manager   | 1       | NULL                                |    2 |   100.00 | Using index; Using join buff
er (hash join)   |
|  1 | SIMPLE      | i     | NULL       | ref    | idx_fk_film_id,idx_store_id_film_id | idx_store_id_film_id | 3       | sakila.s.store_id,sakila.fa.film_id |    3 |   100.00 | Using index
                 |
+----+-------------+-------+------------+--------+--------------------------------------+----------------------+---------+-------------------------------------+------+----------+-----------------------------
-----------------+
5 rows in set, 1 warning (0.00 sec)

mysql> 
```

Visualizing the Query Plan

you can past the query plan in the following webiste MySQL Visual Explain and get more visual query plan for better identify cost in the query plan , follow along the guide and the you will get mush more cleaner query plan



**Identifying Active Sessionsand Locks

Use the SHOW FULL PROCESSLIST command to see what all the connected threads are doing.

```
-- Show all active connections and their queries
SHOW FULL PROCESSLIST;
```

Look at the Time column to see how long a query has been running and the Info column for the query text.
This is your go-to command for seeing "what's running right now."

How to Check for Locks and Deadlocks:

```
SELECT
    r.trx_id AS waiting_trx_id,
    r.trx_mysql_thread_id AS waiting_thread,
    r.trx_query AS waiting_query,
    b.trx_id AS blocking_trx_id,
    b.trx_mysql_thread_id AS blocking_thread,
    b.trx_query AS blocking_query
FROM
    performance_schema.data_lock_waits AS w
JOIN
    information_schema.innodb_trx AS b ON b.trx_id = w.blocking_engine_transaction_id
JOIN
    information_schema.innodb_trx AS r ON r.trx_id = w.requesting_engine_transaction_id;
```

run the query multiable time if no result returned that means there is not blocking.

using percona toolkit

i personally preferred to use perconatoolkit for faster troubleshooting , it will run various script and return to you overall status of MySQL including wait type , hardware metric to identify if there any hardware bottlenecks , and statues of each database with query's statics

install perconatoolkit by downloading the following steps in this link https://docs.percona.com/percona-toolkit/?_gl=1*vj3u9r*_gcl_aw*R0NMLjE3NDkyMzIxNTEuQ2p3S0NBandvNHJDQmhBYkVpd0F4aEpsQ2R5MXN4QjZLTmp1aEVLdnlOZGVuVG9nY

VJDdW9aNHRPQ2stSGJCd2Y3alQ2T29JTTRFZi1ob0NlMUVRQXZEX0J3RQ..*_gcl_au*MjkxMDYwOTA5LjE3NDkyMzIwMzk.*_ga*Nzc2NjM1NDgxLjE3Mzg1MzQ3NzI.*_ga_DXWV0B7PSN*czE3NDk0ODU2NzUkbzMkZzEkdDE3NDk0ODY0NzgkajYwJGwwJGgw

for MySQL we will use pt-query-digest , best common way is to enable slow query log on MySQL either from `my.cnf` for permeate setting up our temporary enable it using `SET GLOBAL` from inside MySQL

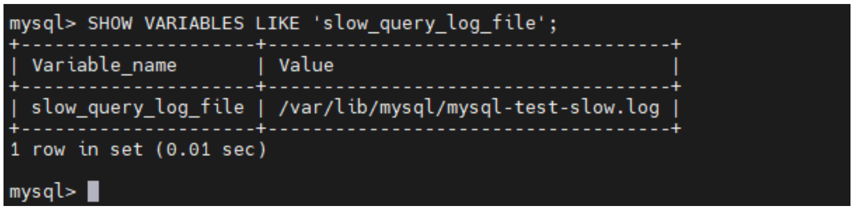
```
mysql -uroot -p

SET GLOBAL slow_query_log = 'ON';
SET GLOBAL log_output = 'FILE';
-- The next line is the most important part

SET GLOBAL long_query_time = 0;
```

`long_query_time = 0`: This tells MySQL to log **every single query**, not just ones that take a long time. This gives `pt-query-digest` a complete picture of your workload.

You can find the location of your slow query log file by running:

```
SHOW VARIABLES LIKE 'slow_query_log_file';
```

The default is often `/var/lib/mysql/your-hostname-slow.log`.

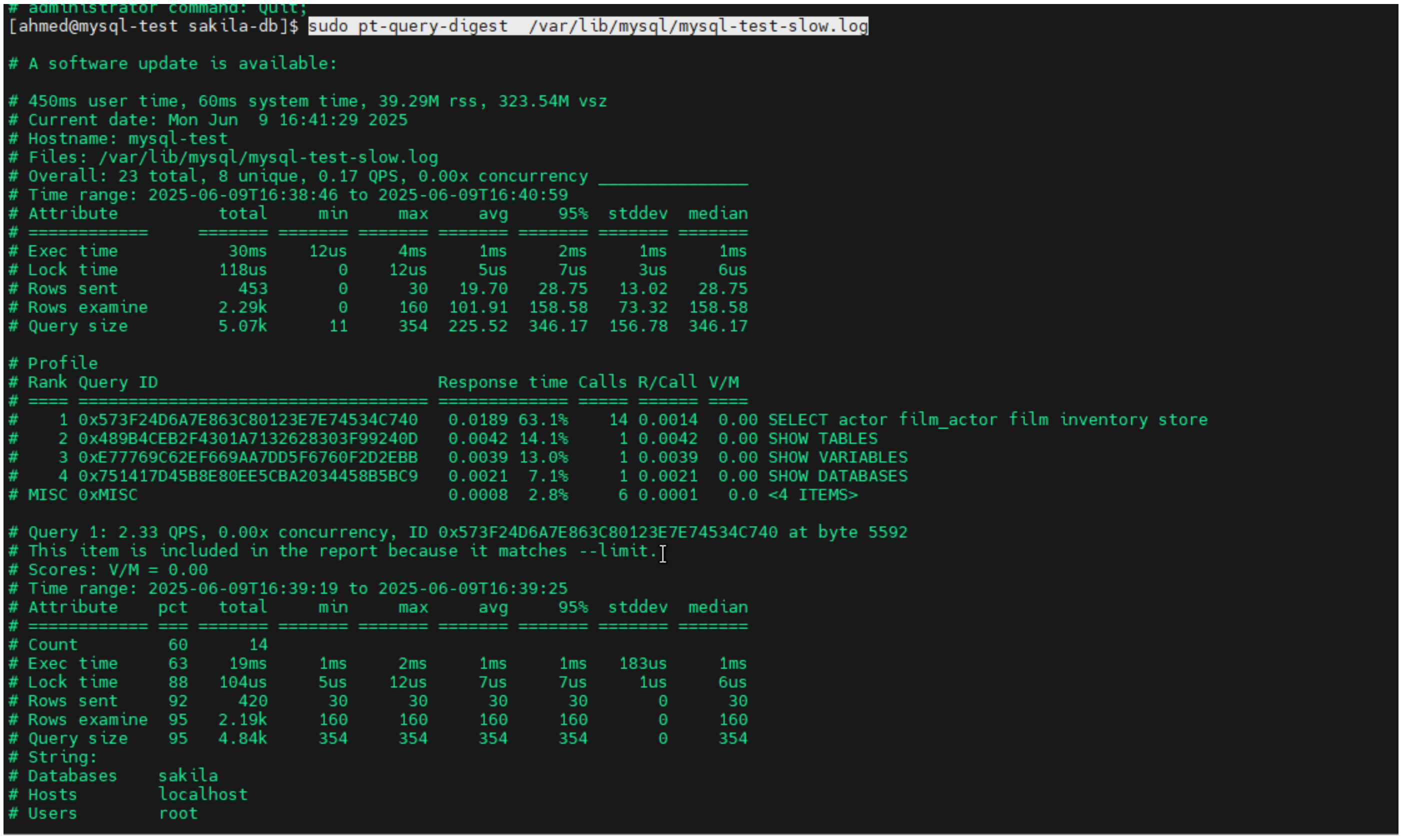


```
mysql> SHOW VARIABLES LIKE 'slow_query_log_file';
+---------------------+-----------------------------------+
| Variable_name       | Value                             |
+---------------------+-----------------------------------+
| slow_query_log_file | /var/lib/mysql/mysql-test-slow.log |
+---------------------+-----------------------------------+
1 row in set (0.01 sec)

mysql> 
```

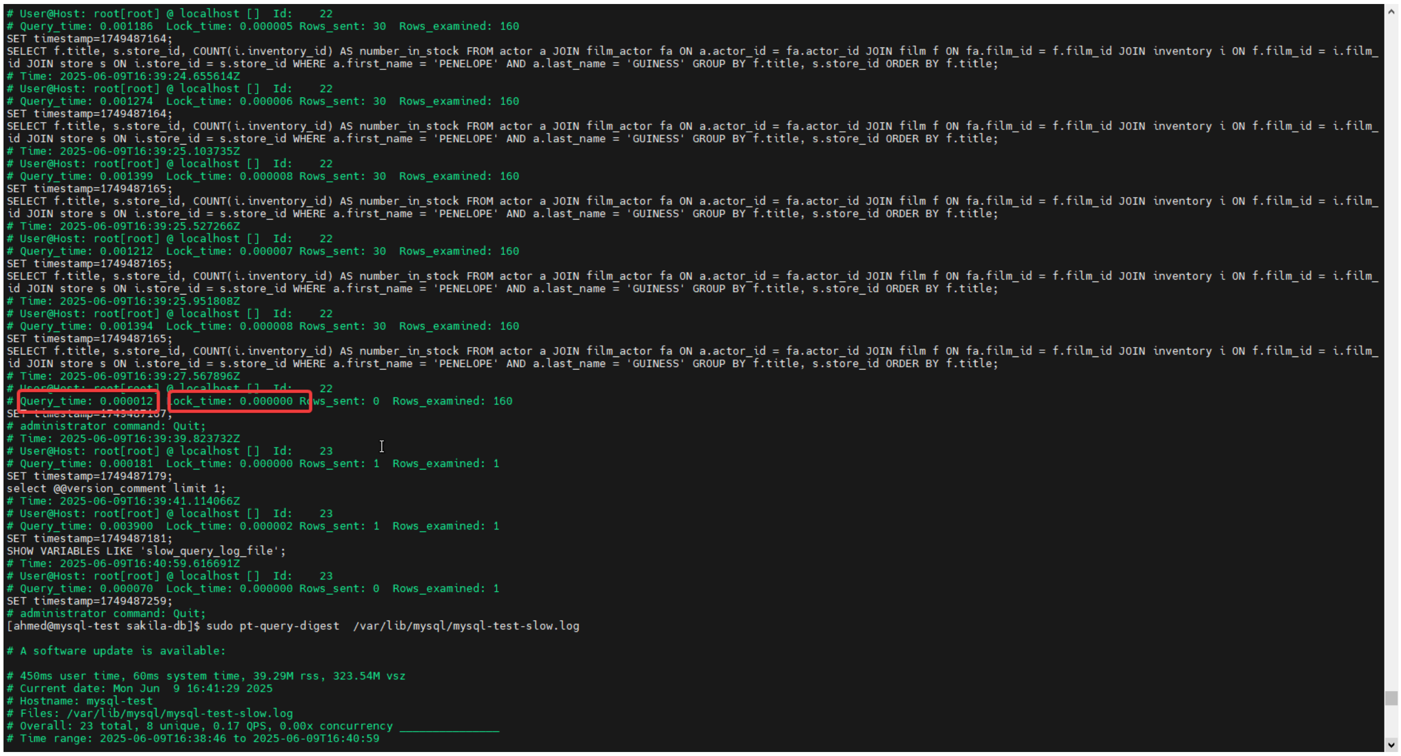
Run `pt-query-digest` This is the simplest and most common use case. It reads the specified log file and prints a detailed report to your screen.

```
sudo pt-query-digest  /var/lib/mysql/mysql-test-slow.log
```

https://www.linkedin.com/in/ahmed-mohamed-423583151



```
# administrator command: Quit;
[ahmed@mysql-test sakila-db]$ sudo pt-query-digest  /var/lib/mysql/mysql-test-slow.log

# A software update is available:

# 450ms user time, 60ms system time, 39.29M rss, 323.54M vsz
# Current date: Mon Jun  9 16:41:29 2025
# Hostname: mysql-test
# Files: /var/lib/mysql/mysql-test-slow.log
# Overall: 23 total, 8 unique, 0.17 QPS, 0.00x concurrency _______________
# Time range: 2025-06-09T16:38:46 to 2025-06-09T16:40:59
# Attribute          total     min     max     avg     95%  stddev  median
# ============     ======= ======= ======= ======= ======= ======= =======
# Exec time           30ms    12us     4ms     1ms     2ms     1ms     1ms
# Lock time          118us       0    12us     5us     7us     3us     6us
# Rows sent            453       0      30   19.70   28.75   13.02   28.75
# Rows examine       2.29k       0     160  101.91  158.58   73.32  158.58
# Query size         5.07k      11     354  225.52  346.17  156.78  346.17

# Profile
# Rank Query ID                            Response time Calls R/Call V/M
# ==== =================================== ============= ===== ====== ====
#    1 0x573F24D6A7E863C80123E7E74534C740   0.0189 63.1%    14 0.0014  0.00 SELECT actor film_actor film inventory store
#    2 0x489B4CEB2F4301A7132628303F99240D   0.0042 14.1%     1 0.0042  0.00 SHOW TABLES
#    3 0xE77769C62EF669AA7DD5F6760F2D2EBB   0.0039 13.0%     1 0.0039  0.00 SHOW VARIABLES
#    4 0x751417D45B8E80EE5CBA2034458B5BC9   0.0021  7.1%     1 0.0021  0.00 SHOW DATABASES
# MISC 0xMISC                               0.0008  2.8%     6 0.0001   0.0 <4 ITEMS>

# Query 1: 2.33 QPS, 0.00x concurrency, ID 0x573F24D6A7E863C80123E7E74534C740 at byte 5592
# This item is included in the report because it matches --limit.
# Scores: V/M = 0.00
# Time range: 2025-06-09T16:39:19 to 2025-06-09T16:39:25
# Attribute    pct   total     min     max     avg     95%  stddev  median
# ============ === ======= ======= ======= ======= ======= ======= =======
# Count         60      14
# Exec time     63    19ms     1ms     2ms     1ms     1ms   183us     1ms
# Lock time     88   104us     5us    12us     7us     7us     1us     6us
# Rows sent     92     420      30      30      30      30       0      30
# Rows examine  95   2.19k     160     160     160     160       0     160
# Query size    95   4.84k     354     354     354     354       0     354
# String:
# Databases    sakila
# Hosts        localhost
# Users        root
```



```
# User@Host: root[root] @ localhost []  Id:    22
# Query_time: 0.001186  Lock_time: 0.000005 Rows_sent: 30  Rows_examined: 160
SET timestamp=1749487164;
SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.film_id = i.film_
id JOIN store s ON i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id ORDER BY f.title;
# Time: 2025-06-09T16:39:24.655614Z
# User@Host: root[root] @ localhost []  Id:    22
# Query_time: 0.001274  Lock_time: 0.000006 Rows_sent: 30  Rows_examined: 160
SET timestamp=1749487164;
SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.film_id = i.film_
id JOIN store s ON i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id ORDER BY f.title;
# Time: 2025-06-09T16:39:25.103735Z
# User@Host: root[root] @ localhost []  Id:    22
# Query_time: 0.001399  Lock_time: 0.000008 Rows_sent: 30  Rows_examined: 160
SET timestamp=1749487165;
SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.film_id = i.film_
id JOIN store s ON i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id ORDER BY f.title;
# Time: 2025-06-09T16:39:25.527266Z
# User@Host: root[root] @ localhost []  Id:    22
# Query_time: 0.001212  Lock_time: 0.000007 Rows_sent: 30  Rows_examined: 160
SET timestamp=1749487165;
SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.film_id = i.film_
id JOIN store s ON i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id ORDER BY f.title;
# Time: 2025-06-09T16:39:25.951808Z
# User@Host: root[root] @ localhost []  Id:    22
# Query_time: 0.001394  Lock_time: 0.000008 Rows_sent: 30  Rows_examined: 160
SET timestamp=1749487165;
SELECT f.title, s.store_id, COUNT(i.inventory_id) AS number_in_stock FROM actor a JOIN film_actor fa ON a.actor_id = fa.actor_id JOIN film f ON fa.film_id = f.film_id JOIN inventory i ON f.film_id = i.film_
id JOIN store s ON i.store_id = s.store_id WHERE a.first_name = 'PENELOPE' AND a.last_name = 'GUINESS' GROUP BY f.title, s.store_id ORDER BY f.title;
# Time: 2025-06-09T16:39:27.567896Z
# User@Host: root[root] @ localhost []  Id:    22
# Query_time: 0.000012  Lock_time: 0.000000 Rows_sent: 0  Rows_examined: 160
SET timestamp=1749487167;
# administrator command: Quit;
# Time: 2025-06-09T16:39:39.823732Z
# User@Host: root[root] @ localhost []  Id:    23
# Query_time: 0.000181  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 1
SET timestamp=1749487179;
select @@version_comment limit 1;
# Time: 2025-06-09T16:39:41.114066Z
# User@Host: root[root] @ localhost []  Id:    23
# Query_time: 0.003900  Lock_time: 0.000002 Rows_sent: 1  Rows_examined: 1
SET timestamp=1749487181;
SHOW VARIABLES LIKE 'slow_query_log_file';
# Time: 2025-06-09T16:40:59.616691Z
# User@Host: root[root] @ localhost []  Id:    23
# Query_time: 0.000070  Lock_time: 0.000000 Rows_sent: 0  Rows_examined: 1
SET timestamp=1749487259;
# administrator command: Quit;
[ahmed@mysql-test sakila-db]$ sudo pt-query-digest  /var/lib/mysql/mysql-test-slow.log

# A software update is available:

# 450ms user time, 60ms system time, 39.29M rss, 323.54M vsz
# Current date: Mon Jun  9 16:41:29 2025
# Hostname: mysql-test
# Files: /var/lib/mysql/mysql-test-slow.log
# Overall: 23 total, 8 unique, 0.17 QPS, 0.00x concurrency _______________
# Time range: 2025-06-09T16:38:46 to 2025-06-09T16:40:59
```

look for query that has high query time or high lock time

remember after you finish using pt-query-digest to change parameter for `long_query_time` from `0` to avoid filling up your disk and adding unnecessary overhead:

```
SET GLOBAL long_query_time = 10; -- Or your previous default
SET GLOBAL slow_query_log = 'OFF';
```

Analyzing the `PROCESSLIST`

This mode repeatedly checks `SHOW FULL PROCESSLIST` for a set duration to see what queries are running right now.

```
pt-query-digest --processlist h=localhost,u=root,p=your_password --run-time 30 > current_activity_report.txt
```

- This will check the process list every second for 30 seconds and create a report.

pt-deadlock-logger

is another essential utility from the **Percona Toolkit**. While `pt-query-digest` is for analyzing general query performance, `pt-deadlock-logger` has a very specific and critical purpose: **to continuously monitor for and create a permanent record of MySQL deadlocks.**

When a deadlock occurs in InnoDB, MySQL resolves it automatically by choosing one transaction as a "victim" and rolling it back. Information about this deadlock is then printed to the output of the `SHOW ENGINE INNODB STATUS` command.

The problem is that this output is **ephemeral**. It only ever shows the **most recent** deadlock detected. If another deadlock occurs five minutes later, the information about the first one is lost forever. This makes it incredibly difficult to debug intermittent deadlock issues.

`pt-deadlock-logger` solves this by running as a background process (a daemon), constantly checking for deadlocks, and when it finds one, it extracts the relevant information and saves it to a file or a database table. This gives you a complete historical log of every deadlock that has occurred.

Interactive Testing (Print to Screen)

This is great for testing your connection. It will check for deadlocks every 5 seconds for a total of one minute and print any findings to your terminal.

```
pt-deadlock-logger --run-time=60s --interval=5s h=localhost,u=pt_user,p=a_very_secure_password
```

https://www.linkedin.com/in/ahmed-mohamed-423583151