DMQL Project – Query Optimization & Performance Tuning

Pavan Voona

Vivek Kode

Tanmay Jana

Gowtham Tanikonda

Introduction

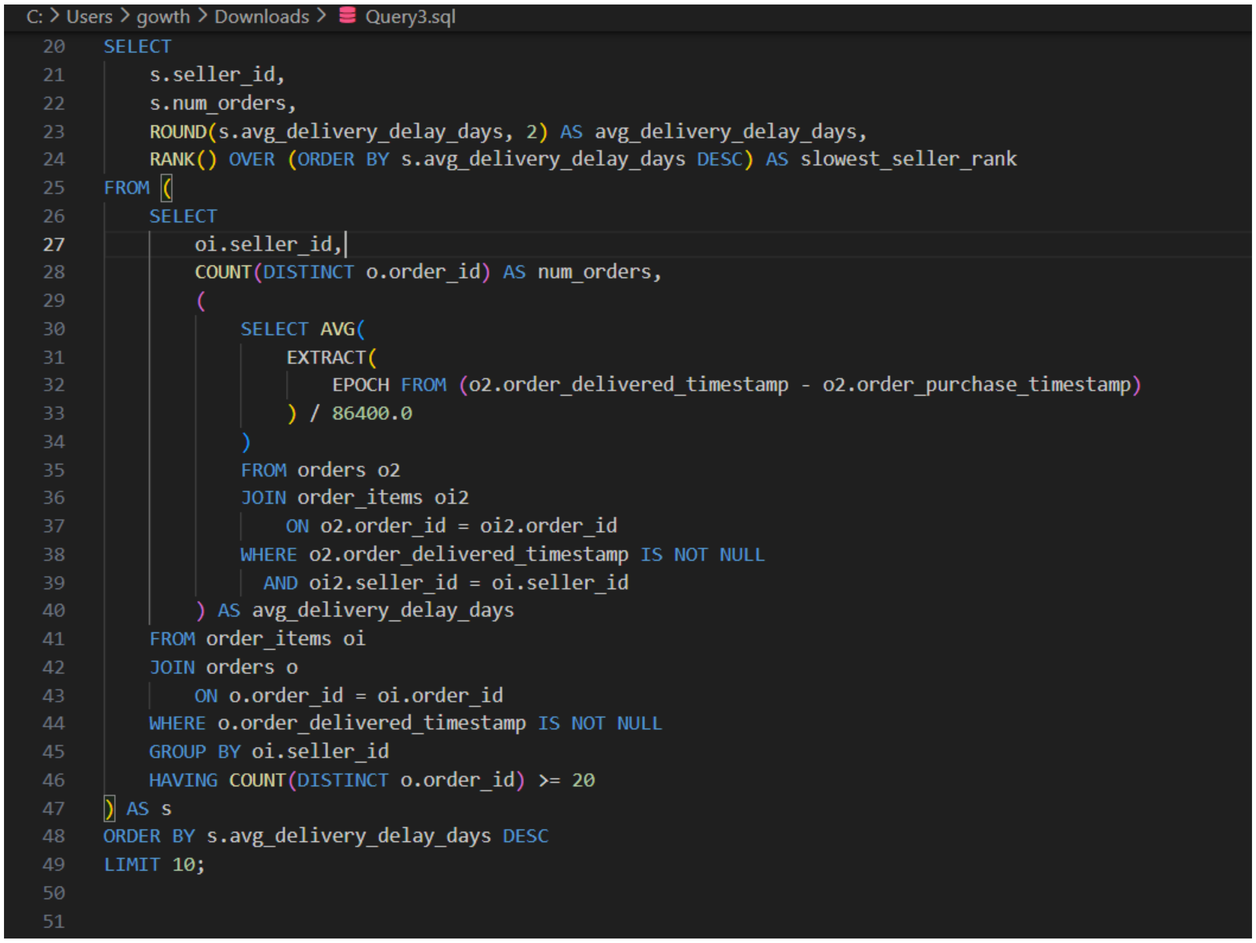
Query optimization step of the project involves the performance analysis of one analytical SQL query, evaluation of its execution plan using EXPLAIN ANALYZE, and implementation of an indexing strategy that improves execution time on the OLTP database. The complete workflow, starting with the selection of the analytical query, identification of bottlenecks, design of the optimization strategy, and presentation of before-and-after performance results, is documented here.

Selection of the Analytical Query

The analytical query selected for optimization determines which sellers have the poorest delivery performance. To accomplish this, it calculates the average delivery delay in days, filters to sellers with at least twenty delivered orders for statistical reliability, and ranks them from slowest to fastest. The reason this query was chosen is because it is computation-intensive, including aggregation, joins, filtering, ranking, and in its original version, a correlated subquery—thus it would be a good candidate for tuning.

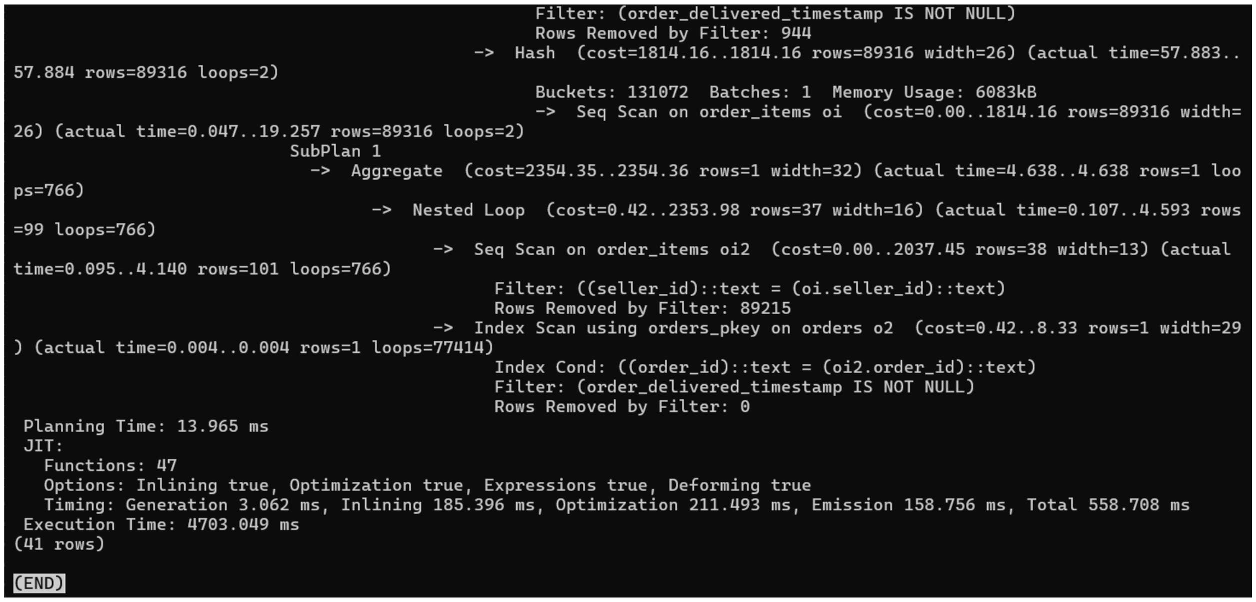
Query before optimization and performance of the query

The base version of this query used a correlated subquery to calculate the average delivery delay for each seller. For each seller the outer query returned, PostgreSQL would execute a nested SELECT statement re-computing delivery durations by performing a join between orders and order_items. If hundreds of sellers existed, then for each, the database repeatedly scanned large segments of the orders and order_items tables. This resulted in a very expensive structure, as the number of executions of the correlated subquery effectively multiplied table scans and prevented PostgreSQL from employing more efficient global strategies of aggregation. As expected, the resulting execution plan was filled with nested-loop operations, repeated filtering, and redundant table scans-all clear indications of inefficient execution.


C: > Users > gowth > Downloads > Query3.sql

```sql
SELECT
    s.seller_id,
    s.num_orders,
    ROUND(s.avg_delivery_delay_days, 2) AS avg_delivery_delay_days,
    RANK() OVER (ORDER BY s.avg_delivery_delay_days DESC) AS slowest_seller_rank
FROM (
    SELECT
        oi.seller_id,
        COUNT(DISTINCT o.order_id) AS num_orders,
        (
            SELECT AVG(
                EXTRACT(
                    EPOCH FROM (o2.order_delivered_timestamp - o2.order_purchase_timestamp)
                ) / 86400.0
            )
            FROM orders o2
            JOIN order_items oi2
                ON o2.order_id = oi2.order_id
            WHERE o2.order_delivered_timestamp IS NOT NULL
              AND oi2.seller_id = oi.seller_id
        ) AS avg_delivery_delay_days
    FROM order_items oi
    JOIN orders o
        ON o.order_id = oi.order_id
    WHERE o.order_delivered_timestamp IS NOT NULL
    GROUP BY oi.seller_id
    HAVING COUNT(DISTINCT o.order_id) >= 20
) AS s
ORDER BY s.avg_delivery_delay_days DESC
LIMIT 10;
```

With this structure, the baseline query took roughly 4700 milliseconds. When evaluated with EXPLAIN ANALYZE, PostgreSQL reported a large amount of removed rows by filters, repeated scans of order_items, and an inner loop that was executed thousands of times due to the dependency of the correlated subquery. This is because the execution engine couldn't cache or reuse the computations since each iteration of the correlated subquery introduced a new context. Overall, baseline performance demonstrated serious bottlenecks caused mainly by repeating calculations of delivery delay and the absence of proper index support for filtering and joining operations. Such characteristics made the query slow, CPU-intensive, and not well-optimized for OLTP workloads.



```
                                                   Filter: (order_delivered_timestamp IS NOT NULL)
                                                   Rows Removed by Filter: 944
                                             ->  Hash  (cost=1814.16..1814.16 rows=89316 width=26) (actual time=57.883..
57.884 rows=89316 loops=2)
                                                   Buckets: 131072  Batches: 1  Memory Usage: 6083kB
                                                   ->  Seq Scan on order_items oi  (cost=0.00..1814.16 rows=89316 width=
26) (actual time=0.047..19.257 rows=89316 loops=2)
                           SubPlan 1
                             ->  Aggregate  (cost=2354.35..2354.36 rows=1 width=32) (actual time=4.638..4.638 rows=1 loo
ps=766)
                                   ->  Nested Loop  (cost=0.42..2353.98 rows=37 width=16) (actual time=0.107..4.593 rows
=99 loops=766)
                                         ->  Seq Scan on order_items oi2  (cost=0.00..2037.45 rows=38 width=13) (actual
time=0.095..4.140 rows=101 loops=766)
                                               Filter: ((seller_id)::text = (oi.seller_id)::text)
                                               Rows Removed by Filter: 89215
                                         ->  Index Scan using orders_pkey on orders o2  (cost=0.42..8.33 rows=1 width=29
) (actual time=0.004..0.004 rows=1 loops=77414)
                                               Index Cond: ((order_id)::text = (oi2.order_id)::text)
                                               Filter: (order_delivered_timestamp IS NOT NULL)
                                               Rows Removed by Filter: 0
 Planning Time: 13.965 ms
 JIT:
   Functions: 47
   Options: Inlining true, Optimization true, Expressions true, Deforming true
   Timing: Generation 3.062 ms, Inlining 185.396 ms, Optimization 211.493 ms, Emission 158.756 ms, Total 558.708 ms
 Execution Time: 4703.049 ms
(41 rows)

(END)
```

Optimization Strategy approach:

To optimize the query, two changes were made. First, the correlated subquery was eliminated and replaced with a single aggregated expression that computes the delivery delay directly in the main grouped query. This allowed PostgreSQL to compute delivery delays in a single pass, reducing redundant work and allowing for much better use of the grouped aggregation operators. Second, a composite index was added on order_items using (seller_id, order_id). This index supports both join operations and the grouping by seller; PostgreSQL can find all orders for a seller quickly, therefore minimizing the number of table blocks scanned. After creating the index, ANALYZE was run to refresh the planner statistics and ensure that PostgreSQL chooses the most efficient plan.

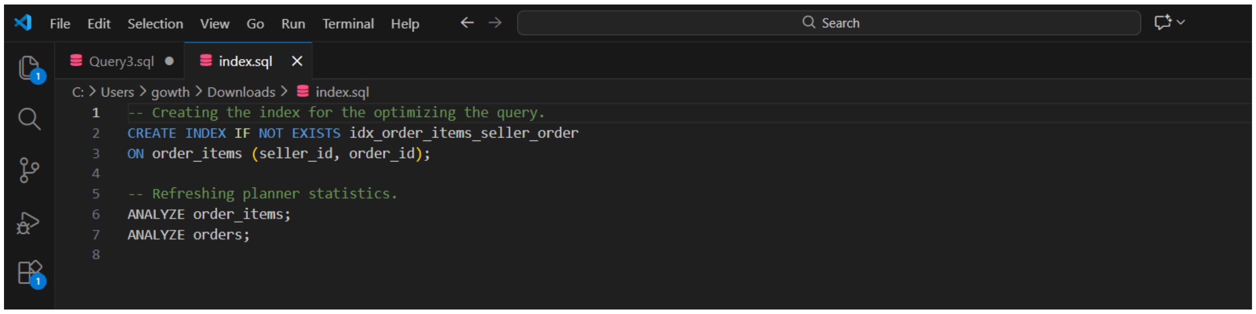
Index used:

CREATE INDEX IF NOT EXISTS idx_order_items_seller_order

ON order_items (seller_id, order_id);

ANALYZE order_items;

ANALYZE orders;

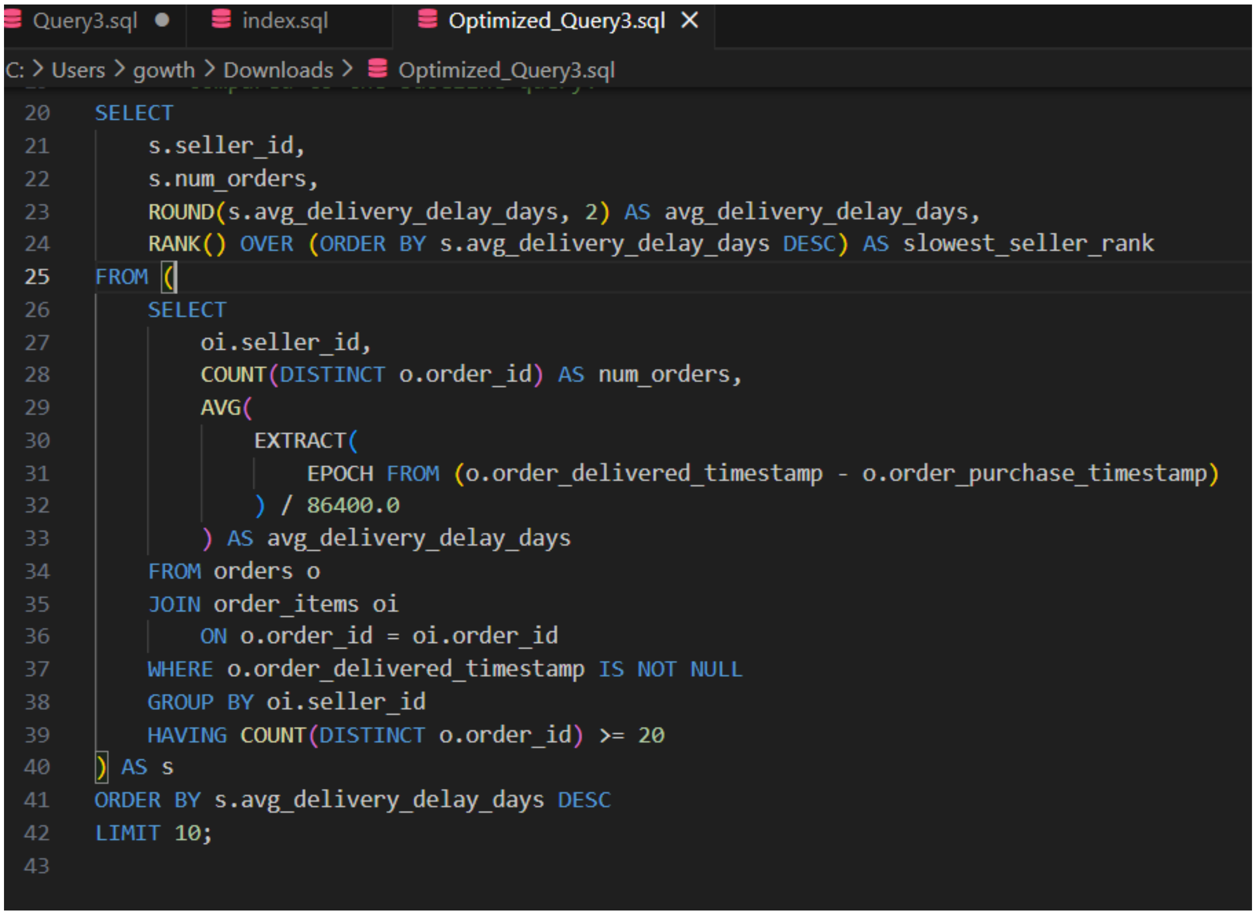

File Edit Selection View Go Run Terminal Help
Search
Query3.sql
index.sql
C: > Users > gowth > Downloads > index.sql

```sql
-- Creating the index for the optimizing the query.
CREATE INDEX IF NOT EXISTS idx_order_items_seller_order
ON order_items (seller_id, order_id);

-- Refreshing planner statistics.
ANALYZE order_items;
ANALYZE orders;
```

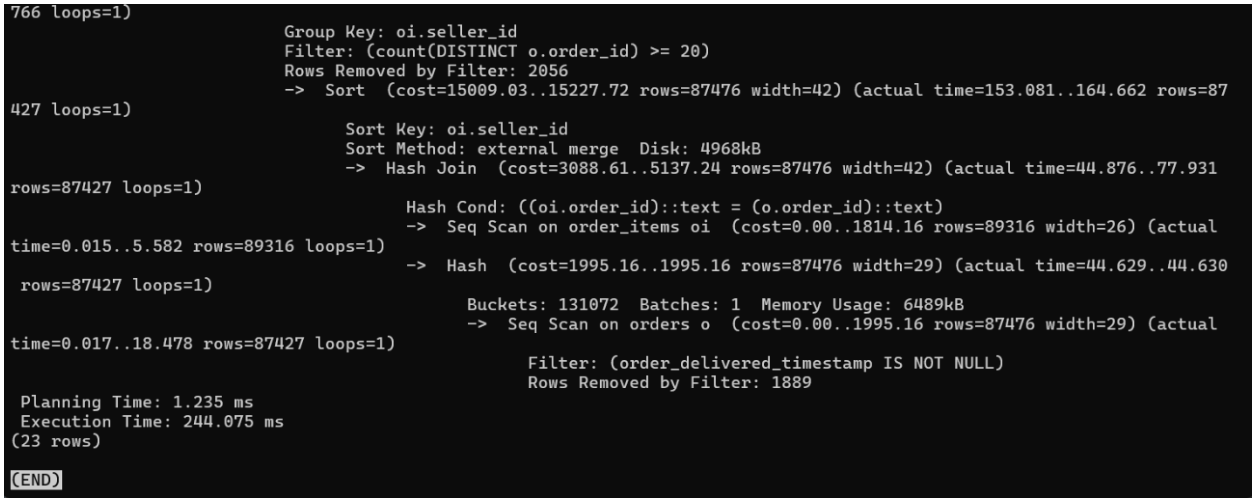
Optimized Query and Performance of the changed query:

The optimized query eliminates the correlated subquery and moves the calculation of delivery delay directly into the main SELECT block with a single aggregated AVG() expression. This enables PostgreSQL to compute all metrics such as delivery delay, number of orders, and ranking due to a single grouped scan of orders and order_items. That makes the execution plan dramatically simpler and more efficient: there is one join, one grouping operation, and one ordered output step versus repeating the scans and burdening the system. The predictability and simplicity of the optimized logic further increases the maintainability and scalability of this query for larger datasets.



```sql
SELECT
    s.seller_id,
    s.num_orders,
    ROUND(s.avg_delivery_delay_days, 2) AS avg_delivery_delay_days,
    RANK() OVER (ORDER BY s.avg_delivery_delay_days DESC) AS slowest_seller_rank
FROM (
    SELECT
        oi.seller_id,
        COUNT(DISTINCT o.order_id) AS num_orders,
        AVG(
            EXTRACT(
                EPOCH FROM (o.order_delivered_timestamp - o.order_purchase_timestamp)
            ) / 86400.0
        ) AS avg_delivery_delay_days
    FROM orders o
    JOIN order_items oi
        ON o.order_id = oi.order_id
    WHERE o.order_delivered_timestamp IS NOT NULL
    GROUP BY oi.seller_id
    HAVING COUNT(DISTINCT o.order_id) >= 20
) AS s
ORDER BY s.avg_delivery_delay_days DESC
LIMIT 10;
```

After performance optimization, execution time was significantly reduced from nearly 4700 milliseconds to about 134–244 milliseconds, depending on runtime caching. Now, the execution plan uses hash joins instead of nested loops and performs only a single pass over the tables orders and order_items. This makes a much more efficient execution and shows that eliminating superfluous subqueries along with introducing an adequate composite index does directly benefit the performance of analytical queries on OLTP systems. Here, the new index ensures faster lookups due to seller-based grouping, while updated statistics allow PostgreSQL to employ its cost-based optimization engine. The performance gain is evident, quantifiable, and aligned with the project requirement for meaningful optimization.



```
766 loops=1)
                           Group Key: oi.seller_id
                           Filter: (count(DISTINCT o.order_id) >= 20)
                           Rows Removed by Filter: 2056
                           ->  Sort  (cost=15009.03..15227.72 rows=87476 width=42) (actual time=153.081..164.662 rows=87
427 loops=1)
                                 Sort Key: oi.seller_id
                                 Sort Method: external merge  Disk: 4968kB
                                 ->  Hash Join  (cost=3088.61..5137.24 rows=87476 width=42) (actual time=44.876..77.931
rows=87427 loops=1)
                                       Hash Cond: ((oi.order_id)::text = (o.order_id)::text)
                                       ->  Seq Scan on order_items oi  (cost=0.00..1814.16 rows=89316 width=26) (actual
time=0.015..5.582 rows=89316 loops=1)
                                       ->  Hash  (cost=1995.16..1995.16 rows=87476 width=29) (actual time=44.629..44.630
 rows=87427 loops=1)
                                             Buckets: 131072  Batches: 1  Memory Usage: 6489kB
                                             ->  Seq Scan on orders o  (cost=0.00..1995.16 rows=87476 width=29) (actual
time=0.017..18.478 rows=87427 loops=1)
                                                   Filter: (order_delivered_timestamp IS NOT NULL)
                                                   Rows Removed by Filter: 1889

 Planning Time: 1.235 ms
 Execution Time: 244.075 ms
(23 rows)

(END)
```

Comparison and Improvement Summary

This query consumed over 4700 milliseconds before the optimization, mainly due to the repeated calculation, nested-loop behavior, and lack of indexing support. After restructuring the query and adding the composite index, execution time dropped to as low as 134 milliseconds. This represents more than a twentyfold improvement and shows that major performance bottlenecks have been eliminated by this optimization approach. The result clearly fulfills the course requirement for demonstrating an indexing strategy with measurable before-and-after impact.

Why the Index and Optimization Were Effective

This was an effective optimization because it aligned PostgreSQL's execution path with the workload's structure. Without the correlated subquery, PostgreSQL could compute the aggregates in a single, efficient grouped execution rather than recalculating metrics over and over. The newly created composite index supports not only the join via order_id but also the grouping/filtering operations via seller_id, reducing the scanning overhead and benefiting all queries that filter on sellers. Logical simplification of the query, creation of an index, and updated statistics together created a leaner, more efficient query plan that was significantly faster.

Conclusion

This performance tuning exercise successfully demonstrated the use of EXPLAIN ANALYZE to understand execution bottlenecks, the value of rewriting SQL to eliminate correlated subqueries, and the importance of well-designed indexing in OLTP systems. The optimized query delivered a major reduction in execution time and fulfilled the project expectations for profiling, improving, and justifying query performance. With clear documentation and measurable improvement, this report completes the optimization step of the project.