Optimizing MySQL Queries with Window Functions and Subqueries

Technologies Used

- MySQL
- MySQL Workbench

1. Introduction

- Modern database-driven applications often face performance challenges when SQL queries are not written efficiently, especially while handling large transactional datasets. Common issues such as correlated subqueries, repeated aggregations, and full table scans can lead to increased execution time and poor scalability.
- This project focuses on identifying and optimizing slow-performing SQL queries within a sales database (`sales_db`). By leveraging modern MySQL features such as window functions and applying appropriate indexing strategies, the queries were redesigned to be more efficient, scalable, and easier to maintain. Performance improvements were systematically validated using `EXPLAIN ANALYZE`, ensuring measurable gains in execution cost, query efficiency, and overall database performance.

2. Objective of the Assignment

The primary objectives of this assignment are:

- To analyze complex and slow-performing SQL queries in a sales database
- To identify key performance bottlenecks, including correlated subqueries and inefficient aggregations
- To optimize query logic using modern MySQL window functions such as RANK() and ROW_NUMBER()
- To improve query performance through appropriate indexing strategies
- To evaluate and compare query execution plans before and after optimization using EXPLAIN ANALYZE

3. Database Overview

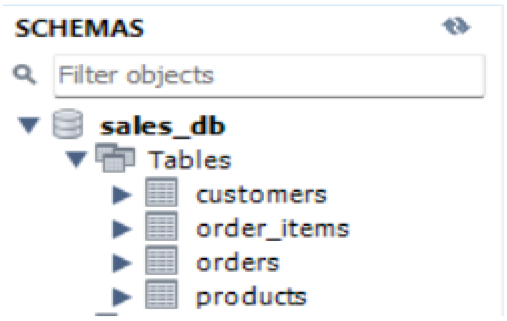


Figure 1 Sales database schema used for query analysis and optimization.

The sales_db database is designed to simulate a real-world sales system and follows a normalized relational structure commonly used in production environments. It captures customer information, product details, order records, and transactional sales data across multiple related tables.

3.1 *Customers Table*

This table stores basic customer information along with their regional details.

- customer_id (Primary Key)
- name
- region

3.2 *Products Table*

This table contains product-related information, including product names and their respective categories.

- product_id (Primary Key)
- product_name
- category

3.3 *Orders Table*

This table maintains order-level details and links customers to their respective orders.

- order_id (Primary Key)
- customer_id (Foreign Key)
- order_date

3.4 *Order_Items Table*

This table captures detailed transactional data for each order, including product quantities and pricing.

- order_item_id (Primary Key)
- order_id (Foreign Key)
- product_id (Foreign Key)
- quantity
- price

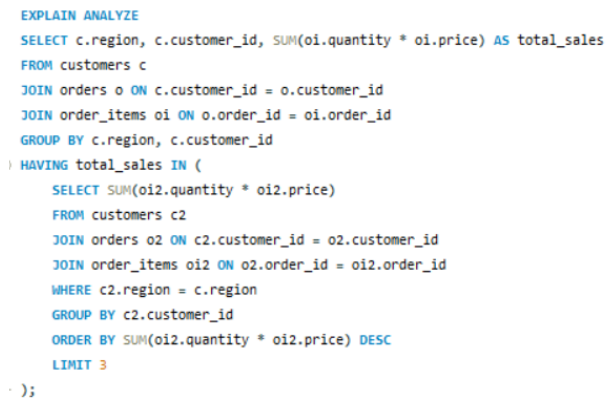
Overall, this schema represents a normalized transactional data model that supports efficient storage, data integrity through foreign keys, and scalable query operations—making it suitable for analyzing and optimizing real-world sales queries.

4. Query 1: Top 3 Customers per Region

4.1 *Business Requirement*

The objective of this query is to identify the **top three customers in each region** based on their **total sales value**. This insight helps businesses understand their most valuable customers across different geographical regions.

4.2 *Original Query* (Slow Approach)



```sql
EXPLAIN ANALYZE
SELECT c.region, c.customer_id, SUM(oi.quantity * oi.price) AS total_sales
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
GROUP BY c.region, c.customer_id
HAVING total_sales IN (
    SELECT SUM(oi2.quantity * oi2.price)
    FROM customers c2
    JOIN orders o2 ON c2.customer_id = o2.customer_id
    JOIN order_items oi2 ON o2.order_id = oi2.order_id
    WHERE c2.region = c.region
    GROUP BY c2.customer_id
    ORDER BY SUM(oi2.quantity * oi2.price) DESC
    LIMIT 3
);
```

Figure 2 Original SQL query used to identify the top 3 customers per region using correlated subqueries and repeated aggregation.

The initial implementation relied on a **correlated subquery** within the HAVING clause to determine the top customers per region.

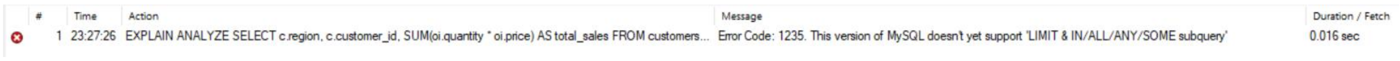
Problems with the Original Query

- Uses correlated subqueries that execute repeatedly for each row
- Recalculates total sales multiple times
- Triggers repeated scans on large tables
- Does not scale well as data volume increases

Performance Analysis

Using EXPLAIN ANALYZE, the execution plan revealed several inefficiencies:

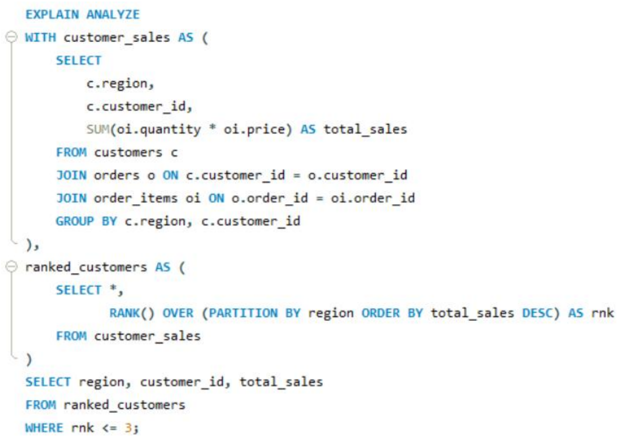
- Presence of dependent subqueries
- Higher overall execution cost
- Large number of rows examined during execution



```
#  Time      Action                                                                                    Message                                                                                         Duration / Fetch
1  23:27:26  EXPLAIN ANALYZE SELECT c.region, c.customer_id, SUM(oi.quantity * oi.price) AS total_sales FROM customers...  Error Code: 1235. This version of MySQL doesn't yet support 'LIMIT & IN/ALL/ANY/SOME subquery'  0.016 sec
```

Figure 3 MySQL execution limitation encountered when attempting to analyze the original correlated subquery containing LIMIT within an IN clause.

4.3 *Optimized Query Using* RANK() *Window Function*



```sql
EXPLAIN ANALYZE
WITH customer_sales AS (
    SELECT
        c.region,
        c.customer_id,
        SUM(oi.quantity * oi.price) AS total_sales
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    JOIN order_items oi ON o.order_id = oi.order_id
    GROUP BY c.region, c.customer_id
),
ranked_customers AS (
    SELECT *,
           RANK() OVER (PARTITION BY region ORDER BY total_sales DESC) AS rnk
    FROM customer_sales
)
SELECT region, customer_id, total_sales
FROM ranked_customers
WHERE rnk <= 3;
```

Figure 4 Optimized query using RANK() window function to rank customers per region.

The optimized solution computes the total sales **only once** and applies the RANK() window function to rank customers **within each region** based on their sales performance.

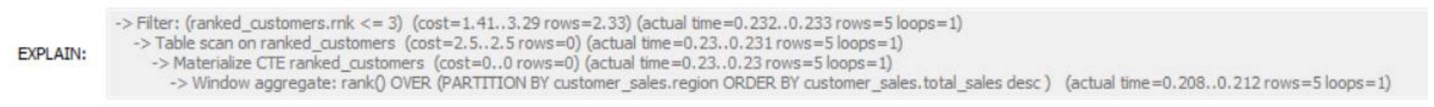
**** Why RANK( ) Was Used ?**

- Specifically suited for **top-N per group** scenarios
- Eliminates the need for correlated subqueries
- Makes the query logic easier to read and maintain
- Scales efficiently with growing datasets

Optimization Strategy

- Aggregate total sales per customer for each region
- Apply RANK() partitioned by region and ordered by total sales
- Filter the results to return only the top three ranked customers per region

Performance Improvement



```
EXPLAIN:  -> Filter: (ranked_customers.rnk <= 3)  (cost=1.41..3.29 rows=2.33) (actual time=0.232..0.233 rows=5 loops=1)
            -> Table scan on ranked_customers  (cost=2.5..2.5 rows=0) (actual time=0.23..0.231 rows=5 loops=1)
              -> Materialize CTE ranked_customers  (cost=0..0 rows=0) (actual time=0.23..0.23 rows=5 loops=1)
                -> Window aggregate: rank() OVER (PARTITION BY customer_sales.region ORDER BY customer_sales.total_sales desc )  (actual time=0.208..0.212 rows=5 loops=1)
```

Figure 5   Execution plan of the optimized query using the RANK() window function, showing efficient filtering (rnk <= 3) and materialized CTE execution.

Validation using EXPLAIN ANALYZE confirmed clear performance gains:

- Single-pass aggregation instead of repeated calculations
- Complete removal of dependent subqueries
- Reduced execution cost
- Improved index utilization and faster execution

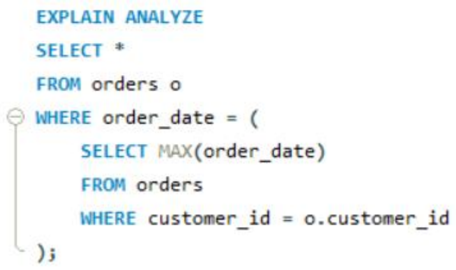
5. <u>Query 2: Latest Order per Customer</u>

5.1 *Business Requirement*

The goal of this query is to retrieve the **most recent order placed by each customer**. This requirement is commonly used in reporting, customer activity tracking, and personalization workflows.

5.2 *Original Query* (Slow Approach)

The initial implementation relied on a **correlated subquery** using MAX(order_date) to identify the latest order for each customer.

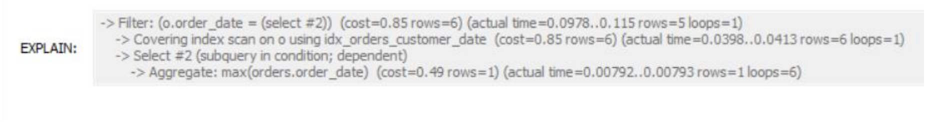


```
EXPLAIN ANALYZE
SELECT *
FROM orders o
WHERE order_date = (
    SELECT MAX(order_date)
    FROM orders
    WHERE customer_id = o.customer_id
);
```

Figure 6 Original query using correlated subquery to fetch latest order per customer.

Problems with the Original Query

- Executes the subquery repeatedly for each customer record
- Performs poorly on large order tables
- Does not scale well as data volume grows

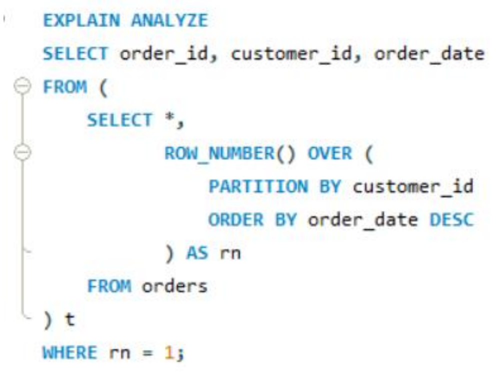


```
EXPLAIN:  -> Filter: (o.order_date = (select #2))  (cost=0.85 rows=6) (actual time=0.0978..0.115 rows=5 loops=1)
            -> Covering index scan on o using idx_orders_customer_date  (cost=0.85 rows=6) (actual time=0.0398..0.0413 rows=6 loops=1)
            -> Select #2 (subquery in condition; dependent)
              -> Aggregate: max(orders.order_date)  (cost=0.49 rows=1) (actual time=0.00792..0.00793 rows=1 loops=6)
```

Figure 7 Execution plan showing repeated subquery execution.

5.3 *Optimized Query Using* ROW_NUMBER()

The optimized solution leverages the ROW_NUMBER() window function to rank orders **per customer** based on the order date.



```
EXPLAIN ANALYZE
SELECT order_id, customer_id, order_date
FROM (
    SELECT *,
           ROW_NUMBER() OVER (
               PARTITION BY customer_id
               ORDER BY order_date DESC
           ) AS rn
    FROM orders
) t
WHERE rn = 1;
```

Figure 8 Optimized query using ROW_NUMBER() to efficiently retrieve latest order per customer.

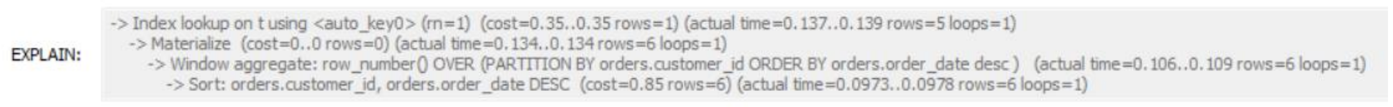
** Why ROW_NUMBER( ) Was Used ?

- Ideal for identifying the **latest record per group**
- Completely removes the need for correlated subqueries
- Ensures deterministic and predictable ordering
- Executes efficiently in a single pass over the data

Optimization Strategy

- Partition the dataset by customer_id
- Order records by order_date in descending order
- Select only rows where ROW_NUMBER() = 1

Performance Improvement



```
EXPLAIN:  -> Index lookup on t using <auto_key0> (rn=1)  (cost=0.35..0.35 rows=1) (actual time=0.137..0.139 rows=5 loops=1)
            -> Materialize  (cost=0..0 rows=0) (actual time=0.134..0.134 rows=6 loops=1)
              -> Window aggregate: row_number() OVER (PARTITION BY orders.customer_id ORDER BY orders.order_date desc )  (actual time=0.106..0.109 rows=6 loops=1)
                -> Sort: orders.customer_id, orders.order_date DESC  (cost=0.85 rows=6) (actual time=0.0973..0.0978 rows=6 loops=1)
```

Figure 9 Execution plan showing elimination of correlated subqueries and improved efficiency.

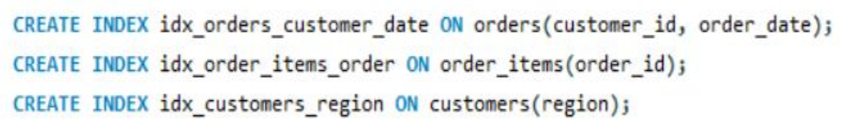
Validation using EXPLAIN ANALYZE demonstrated significant improvements:

- Elimination of dependent subqueries
- Reduced execution time
- Cleaner and more efficient execution plan

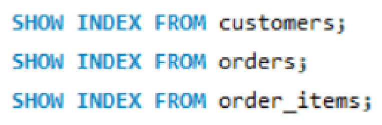
6. Index Optimization

In addition to restructuring queries, targeted indexing strategies were applied to further improve data access speed and overall query performance. These indexes were designed to support frequent joins, filtering, and window function operations.

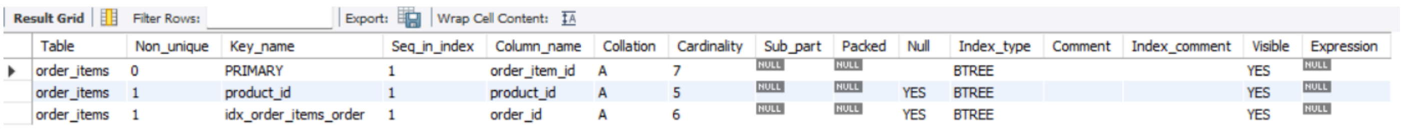
6.1 *Indexes Created*



```sql
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
CREATE INDEX idx_order_items_order ON order_items(order_id);
CREATE INDEX idx_customers_region ON customers(region);
```



```sql
SHOW INDEX FROM customers;
SHOW INDEX FROM orders;
SHOW INDEX FROM order_items;
```



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
order_items	0	PRIMARY	1	order_item_id	A	7	NULL	NULL		BTREE			YES	NULL
order_items	1	product_id	1	product_id	A	5	NULL	NULL	YES	BTREE			YES	NULL
order_items	1	idx_order_items_order	1	order_id	A	6	NULL	NULL	YES	BTREE			YES	NULL

Figure 10 Indexes created to optimize joins, filtering, and window function execution.

6.2 *Why These Indexes Were Used ?*

- **orders(customer_id, order_date)**
  Supports efficient partitioning and ordering required by the ROW_NUMBER() window function when retrieving the latest order per customer.
- **order_items(order_id)**
  Improves join performance between the orders and order_items tables by reducing lookup time.
- **customers(region)**
  Enhances region-based grouping and filtering used in analytical queries such as identifying top customers per region.

Impact of Index Optimization

- Significant reduction in full table scans
- Faster join execution
- Lower overall execution cost

7. Execution Plan Comparison

Metric	Original Query	Optimized Query
Subqueries	Correlated	Eliminated
Aggregation	Repeated	Single-pass
Index Usage	Limited	Effective
Execution Cost	High	Reduced
Scalability	Poor	High

This comparison clearly highlights the performance gains achieved through query restructuring and indexing.

8. Tools Used for Validation

The following tools were used to validate and measure performance improvements:

- **MySQL Workbench**
  Used for query execution, optimization, and execution plan analysis.
- **EXPLAIN ANALYZE**
  - Measured actual execution time
  - Verified index usage
  - Compared performance before and after optimization

9. Conclusion

This Assignment demonstrates how modern MySQL features, particularly window functions and indexing, can significantly enhance query performance. By eliminating correlated subqueries, minimizing redundant computations, and validating improvements using EXPLAIN ANALYZE, the optimized queries achieved better efficiency, scalability, and maintainability.

The solutions presented reflect practical, real-world database optimization techniques commonly used in production environments.

10. Key Learnings

- Window functions are powerful tools for solving analytical and grouping problems efficiently
- EXPLAIN ANALYZE is essential for validating and understanding query performance improvements
- Indexing works best when combined with optimized query design
- Clean and structured SQL improves both performance and long-term maintainability

Submitted by:
BIBHUTI JAIPURIA
721MM1031(NIT ROURKELA)
+91-9583432228