Journal of Informatics and Web Engineering
Vol. 5 No. 3 (October 2026) eISSN: 2821-370X

SQL Optimization in e-Commerce Platforms Using Large Language Models: A Multi-Pattern Query Empirical Study

Min Ming Lee[1], Nabhan D. Salih[2*]
[1]Sunway University, No. 5, Jalan Universiti, Bandar Sunway, 47500 Selangor, Malaysia
[2]Department of Computer Science and Engineering, Ohio State University, Columbus, OH 43210-1210, USA.
**corresponding author: (salih.43@osu.edu; ORCID: 0000-0001-7906-4071)*

Abstract - The rapid expansion in e-commerce domains has presented an unprecedented challenge to database systems that support the execution of complex analytic queries on multi-billion transaction data stream while keeping queries' response latency under one second. It is commonly observed that Large Language Models (LLMs) are very good at processing natural language queries, rather than struggling with syntax or semantics. This feature is instrumental to various tasks like analytics acceleration, transaction speed-up and handling large workloads across different domains. In this paper, a detailed examination of how LLMs can be employed as intelligent assistants for Structure Query Language (SQL) query optimization throughout eight different types of query patterns is performed (with two test cases each, resulting in 16 test cases). An in-depth experimental study is carried out among three main database systems (PostgreSQL 16, MariaDB ColumnStore, and SQLite) with relevant real-world e-commerce datasets such as Brazilian E-Commerce Public Dataset and Synthetic e-Commerce Relational Dataset ranging from 10 million to 1 billion records. Our methodology of sixteen test-cases includes temporal aggregations, multi-dimensional grouping, ranking queries, window functions, complex joins, full-text search, recursive queries and geospatial analysis. Results demonstrate that implementing the LLM's recommended query, indexing, and schema-level changes yielded performance improvements ranging from 43% to 94% across test cases, whereby temporal aggregations yield 89% average improvement while multi-dimensional grouping nets 86% improvement on the whole system.

Keywords—Large Language Model, Database Optimization, Query Performance, e-Commerce, Structured Query Language, Index Selection, Query Patterns

Received: 14 June 2025; Accepted: 23 January 2026; Published: 16 October 2026

This is an open access article under the CC BY-NC-ND 4.0 license.

1. INTRODUCTION

Modern e-commerce websites handle an enormous number of transactions per day and need to be able to answer hundreds of analytical queries at the same time for BI, anti-fraud, inventory optimization and personal recommendation purposes [1], [2], [3]. The analytical workloads in question are complex multi-table joins,

Journal of Informatics and Web Engineering
https://doi.org/10.33093/jiwe.2026.5.3.8
© Universiti Telekom Sdn Bhd.
Published by MMU Press. URL: https://journals.mmupress.com/jiwe

extensive aggregations over temporal dimensions and sophisticated filtering across high-cardinality categorical attributes [4].

In particular, the leading database management systems use cost-based query optimizers [5], [6] which are based on mathematical models for estimating execution costs. Nonetheless, it is a big problem for these optimizers to estimate cardinality, there is also limitation in search space exploration, statistics become old and it is impossible to take advantage of domain-specific query patterns [7]. In this paper, we want to find out if the Large Language Models (LLMs) can act as more effective decision support systems when it comes to database tuning, adding learned heuristics to the traditional optimizers [8], [9]. For this purpose, we define eight query pattern categories with two representative test cases for each category, making a total of 16 comprehensive case studies.

2. LITERATURE REVIEW

The main idea behind query optimization is to translate declarative Structured Query Language (SQL) queries into executable execution plans. These kinds of optimizers are called cost-based and they started with the System R project [10]. They first generate candidate execution plans and then determine their costs using statistical models before they finally select the plan the has minimum estimated cost. As we mentioned before N-table joins elicits a very large search space (N! possible join orders), for this reason, sophisticated pruning strategies such as dynamic programming [11] and heuristic searching have been developed [12].

Cardinality estimation is central to the query optimization—the prediction of how many rows each operator in an execution plan will produce [13]. The old methods used statistical synopses such as histograms, sampling and independence assumptions [14]. Nevertheless, these techniques have some basic restrictions in case of correlated attributes, multi-column predicates and complex filtering conditions [15]. According to Lohman cardinality estimation errors are increasing in the geometric progression through the query plans taking 10% as a base cardinality estimate may result in 100x error in final cost estimates [16]. As it appeared recently about 50-60% of all queries optimization problems which occur on production system are due to bad cardinality estimates [17].

On the other hand, machine learning has been successfully integrated into database systems over the last ten years; this is done with methods tailored to different parts of the query processing pipeline. Learned cardinality estimation is a hot research topic, and neural network-based estimators come to an order-of-magnitude less errorwise than traditional histogram methods [18]. The deep autoregressive models of Kipf et al. exploit queries execution history for modeling complex joint distributions better than commercial optimizer estimates workload characteristics on benchmark workloads [19].

There were some notable successes in using reinforcement learning for the joint order optimization which views the issue as a learning process. Marcus et al. proposed deep Q-learning to learn the joint orders, and they can improve the performance of the best commercial optimizer up to 2.8x on complex analytical queries [20]. Similarly, the Deep Q-Network (DQN)-based methods have also demonstrated remarkable performance for the different scales of database and complexities of query with a good degree of consistency [21]. Rather than the B-tree schemes, learned index structures do model the data distribution, which leads to up to 3x speedup or point queries and a 99% reduction in space consumption for some workloads[21]. Nonetheless, these techniques still suffer from some practical problems: they require lots of particular training data – typically millions of query executions, have low generalization to new query patterns, and have difficulties with dynamic workloads where data distributions change over time [22]. They are not recommended in heterogeneous enterprise environments because there is a need to retrain them in each new database instance [23], [24].

On the other hand, LLMs show remarkable performance [25], [26] in a variety of database tasks. For instance, GPT-4 and CodeLlama models perform well with an execution accuracy rate ranging from 80% to 85% on complex benchmarks such as Spider and WikiSQL [27], [28]. These systems convert natural language queries into syntactically correct SQL statements which reduce the technical barriers for non-technical users who want to access databases. Additionally, schema design is also emerging with LLMs providing suggestions for a normalization process, issuance of foreign keys, and indexing strategies based on workload [29]. Yet, most of the previous work on this area is more concerned with query correctness rather than on query performance. The text to SQL systems focuses on semantic accuracy, which is execution efficiency of generated queries and it does not consider index utilization or scalability [25], [26]. Trummer et al. [30] have recently published a work on learned query rewriting that highlights the optimization potential of LLM; however, their approach (CodexDB) focuses on generating

custom processing code for query execution from natural language instructions, rather than recommending index, schema, or rewrite-level optimizations against the underlying DBMS as this work does .

Some of the gaps remain unaddressed in existing literature are as follows.

1) Limited Pattern Coverage: Most existing studies focused on specific types of queries to assess LLM optimization (only SELECT-FROM-WHERE clause), which hinder evaluation of its performance on more advanced queries such as Common Table Expresssion (CTE), window functions, or recursive queries [32].
2) Single-Database Evaluation: Most existing studies focus on one DBMS (e.g. PostgreSQL) which gives little knowledge about generalization between various databases. This is an important thing to consider since features are heterogeneous across systems [33].
3) Small-Scale Validation: Prior work has so far been tested only on the datasets of the order of hundreds of thousands to low millions of rows, hence scalability to billion-row datasets remains an untouched area [34] .

Despite the absence of a systematic taxonomic structure, existing research has not comprehensively evaluated LLM-assisted SQL optimization across diverse e-commerce query patterns and multiple database engines. To address these gaps, this study proposes a structured evaluation comprising 16 test cases across eight query-pattern categories, three database systems (PostgreSQL, MariaDB ColumnStore, and SQLite), and dataset scales ranging from 10 million to 1 billion records. The study systematically evaluates the performance effects of LLM-generated recommendations involving query rewriting, index selection, and structural database optimization. The evaluation provides a systematic assessment of LLM-assisted query optimization across diverse e-commerce workloads and database environments.

3. LLM-ASSISTED OPTIMIZATION FRAMEWORK

This section introduces our generic framework to use LLM as an intelligent assistant for improving SQL queries. We explain here the architectural components, operational workflow, prompt engineering strategies and implementation considering that support effective LLM guided optimization across various database systems and query patterns.

Figure 1 depicts the proposed LLM-Assisted Optimization Framework. The proposed system is designed to be independent from DBMS at the level of the framework, yet it is able to recommend various optimization techniques that can differ from one DBMS to another; thus, the optimization techniques that can be recommended by the framework, such as creating materialized views, building GIN or GiST indexes, can vary across different DBMSs, which is elaborated further in the following sections of this paper.

The schema of the framework consists of four modules: (1) Query Analysis Module which recognizes patterns and extracts schemas, (2) LLM Recommendation Engine which generates strategies for optimization, (3) Optimization Strategy Module that validates and estimates cost of implementation, and (4) Execution & Feedback Module that assesses the performance. In the first part, the Query Analysis Module accepts raw SQL queries as input and performs various operations on them, including analyzing their structure, identifying relationships on various queries, and extracting relevant schema data. This module serves as a pre-processing phase for the LLM, preparing the ground for its subsequent actions. The LLM Recommendation Engine, which constitutes the second stage, then exploits the information acquired in the first stage to provide hints ranging from simple re-index recommendation to more involved rewrites of the query structures or schema reorganization. In the Optimization Strategy Module of stage three, the hints provided by the LLM are evaluated in terms of their compliance with database constraints, estimated for implementation costs, and ranked accordingly. Finally, in the last stage, the Execution & Feedback Module captures the chosen optimization strategies, executes the queries in constrained environments, measures the degree of improvement, and relays feedback to the preceding stages.

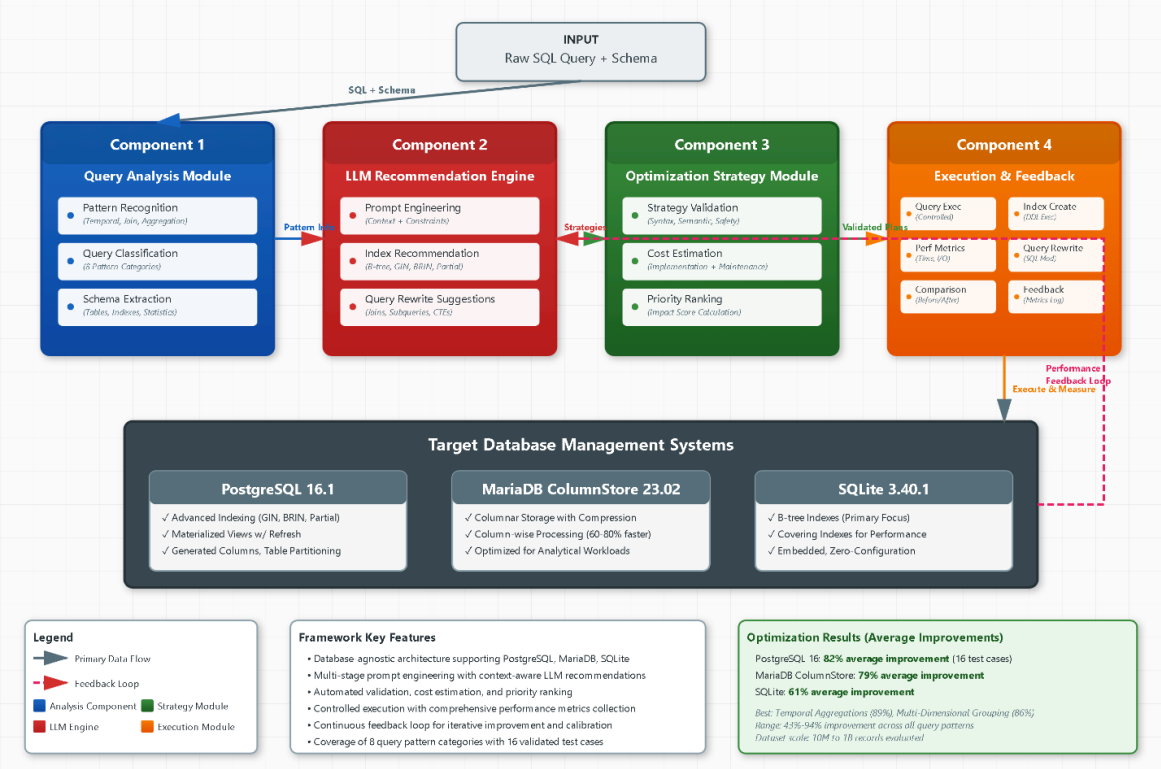


Figure 1. LLM-Assisted Optimization Framework Architecture

3.1 Query Analysis Module

The query is generally grouped into 8 groups as depicted in Table 1.

Table 1. Query Class and Its Associated SQL Clauses

Query Class	**Typical SQL clauses**
Temporal Operations	Detection of date/time functions (YEAR(), MONTH(), DATE_TRUNC()), temporal predicates, and time-series aggregations
Aggregation Patterns	Identification of GROUP BY cardinality (single vs. multi-dimensional), aggregate function types (SUM, COUNT, AVG), and HAVING clauses
Join Structures	Analysis of join types (INNER, LEFT, RIGHT, FULL), join cardinality (one-to-one, one-to-many, many-to-many), and join graph complexity
Subquery Classification	Recognition of correlated vs. uncorrelated subqueries, scalar vs. table subqueries, and EXISTS/IN predicates
Window Function Usage	Detection of partitioning schemes, ordering requirements, and frame specifications
Full-Text Requirements	Identification of LIKE/ILIKE patterns, regular expressions, and text search operations
Recursive Structures	Recognition of WITH RECURSIVE clauses, hierarchical relationships, and graph traversal patterns
Geospatial Operations	Detection of spatial functions (ST_Distance, ST_Contains), coordinate-based filtering, and spatial joins

The module extracts comprehensive schema metadata to inform optimization decisions:

- Table Statistics: Row counts, table size, column cardinalities, data distribution skew
- Existing Indexes: Index types (B-tree, Hash, Generalized Inverted Index (GIN), Block Range Index (BRIN)), indexed columns, index size, usage statistics
- Column Metadata: Data types, NULL percentage, uniqueness, foreign key relationships
- Constraint Information: Primary keys, foreign keys, unique constraints, check constraints
- Partitioning Scheme: Partition type (range, list, hash), partition key, number of partitions

This metadata is serialized into a structured JSON format for LLM consumption, ensuring consistent representation across query patterns and database systems.

3.2 LLM Recommendation Engine

The LLM Recommendation Engine is the cognitive heart of the framework and its task is to develop optimization strategies specifically for different contexts. GPT-4 (gpt-4-0125-preview) with temperature=0.2 is used in our implementation to find a balance between creative suggestions and completely deterministic recommendations.

A multi-stage prompting approach is employed to maximize recommendation quality is shown in Figure 2.

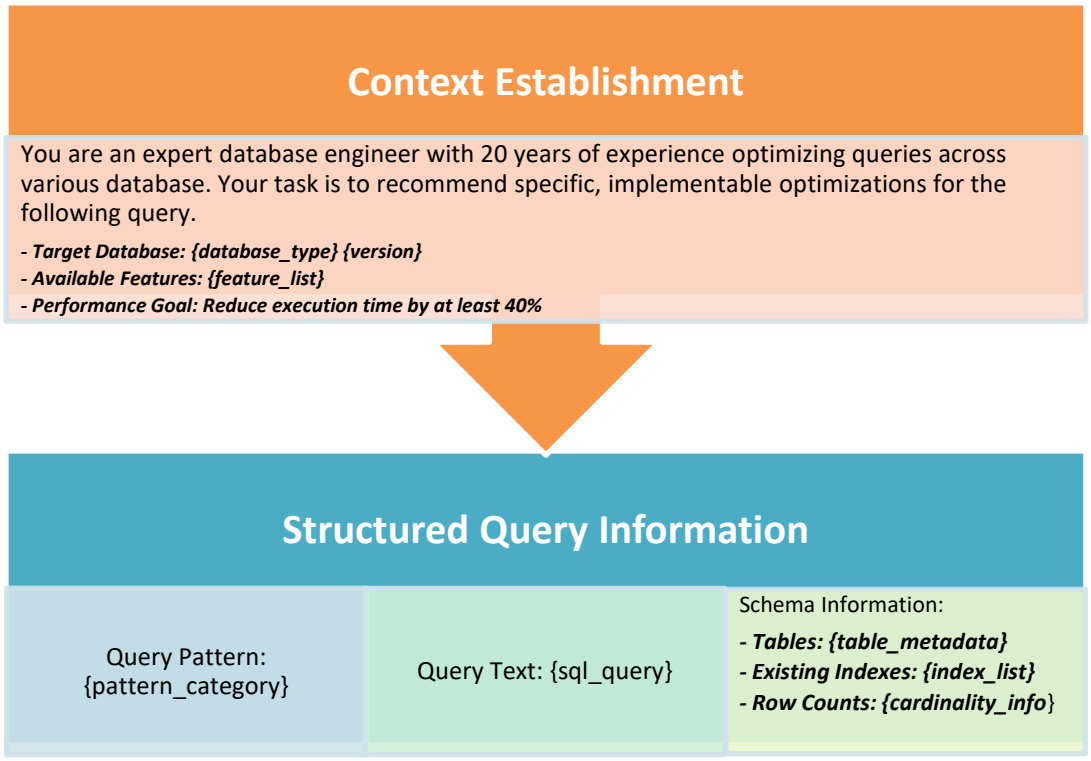


Figure 2. Stages in LLM Recommendation Engine

The LLM generates index recommendations following database-specific best practices (see Figure 3). Besides indexing, the LLM uses query rewrite techniques such as move selection (putting filters close to any incoming scan), join redundancy elimination (filtering irrelevant tables), uncorrelated subquery relations (transforming a correlated

subquery into a join), and optimization of CTEs. For each of these transformations, there are examples in SQL to show the transformation of the SQL from one state to another.

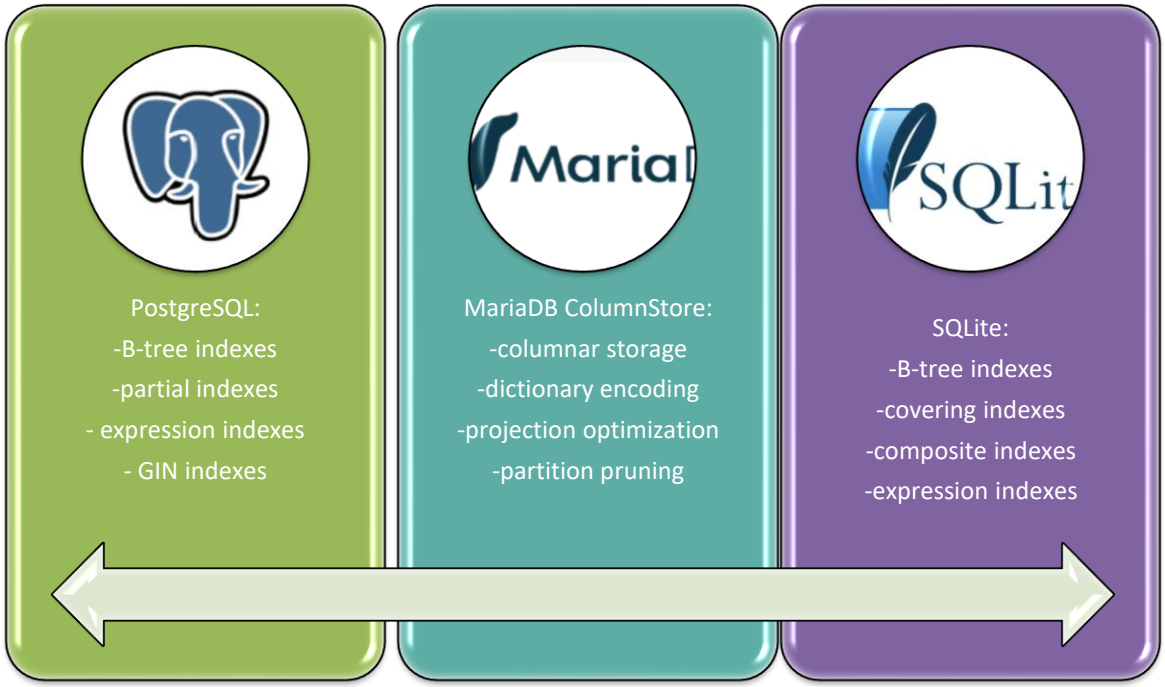


Figure 3. Index Recommendation

3.3 Optimization Strategy Module

The proposed structure encompasses the validation process that occurs in several stages and a decision-making mechanism for recommendations of optimization, which are generated by LLM. Each single piece of advice goes through first syntactic, semantic, constraint, and safety validation checks to confirm its accuracy, compatibility, and safety. Validated recommendations are then analyzed along implementation/maintenance/storage cost dimensions which are later summed with performance improvement estimates for priority score computation. Based on this score, recommendations are either automatically approved for testing, flagged for manual review, or deferred for future consideration.

3.4 Execution and Feedback

The execution and feedback component is in a very specific controlled environment to ensure a dependable assessment of performance. The results from performance are feedback by immediate alerts for unexpected behavior into LLM and also by learning statistics, thus supporting iterative improvement of future recommendations.

4. EXPERIMENTAL DESIGN

The evaluations were conducted in a single-server configuration based on Intel Xeon 16-core processor with 64GB of RAM and 1TB SSD storage on Ubuntu 22.04 LTS operating system. To avoid inconsistency in measurements, there was no other load running during the evaluation. The specific database configurations used included PostgreSQL 16: shared_buffers=8GB work_mem=64MB, MariaDB ColumnStore: innodb_buffer_pool_size=16GB, and SQLite: default analytical configuration.

4.1 Datasets

Two datasets were employed in the experimental evaluation as follows:

- Brazilian E-Commerce Public Dataset: 100K base orders, synthetic scaling to 10M, 50M, 100M, 500M, 1B rows [37].
- Synthetic E-Commerce Relational Dataset: Normalized schema with customers, products, orders, reviews [38].

The synthetic generation maintains the realistic distributions: the temporal patterns (seasonal variations), the geographic correlations (regional buying patterns), and the product-price relationships.

4.2 Database Systems Configuration

Figure 4 shows the configuration of the three selected database systems for the experimental evaluation.

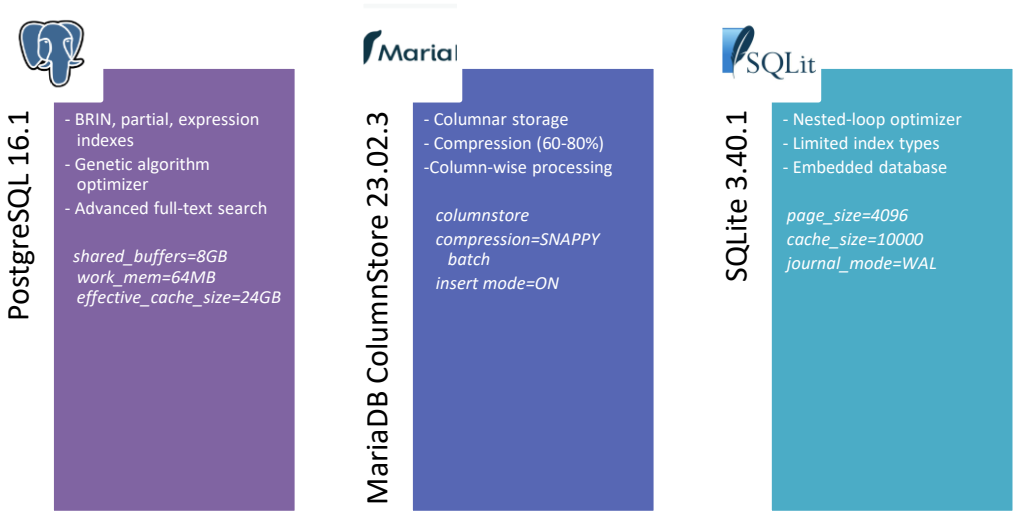


Figure 4. Key Features and Configuration on the Selected Database Systems

4.2 Query Pattern Taxonomy

The evaluation approach organizes test scenarios around 8 query pattern categories, each depicted through 2 test cases (see Table 2 for the complete list). Improvement percentages are rounded to integers for readability. In case functionalities of a database do not support a feature (e.g., partial indexes, GIN, materialized views), then equivalent logical optimizations or N/A results are provided.

Table 2. Query Pattern Taxonomy and Test Case Distribution

Category	Test Cases	Description
Temporal Aggregations	2	Monthly revenue, year-over-year growth
Multi-Dimensional Grouping	2	Region × Category, Segment × Period
Top-N Ranking	2	Best customers, top products per category
Window Functions	2	Running totals, moving averages
Complex Joins	2	5-table reports, correlated subqueries
Full-Text Search	2	Product search, review analysis
Recursive Queries	2	Category hierarchy, bill of materials
Geospatial Analysis	2	Regional sales, shipping distance

4.2.1 Temporal Aggregation (Test Cases T1 and T2)

Temporal aggregations analyze data across time dimensions, critical for trend analysis and business intelligence.

Test Case T1: Monthly Revenue by Category

```
SELECT YEAR(order_date) year, MONTH(order_date) month, category, SUM(revenue) total_revenue FROM orders WHERE status = 'Shipped' GROUP BY YEAR(order_date), MONTH(order_date), category ORDER BY total_revenue DESC;
```

LLM Recommendation: Create generated columns for YEAR() and MONTH() extractions to enable index usage.

```
ALTER TABLE orders ADD COLUMN order_year INT GENERATED ALWAYS AS (YEAR(order_date)), ADD COLUMN order_month INT GENERATED ALWAYS AS (MONTH(order_date)); CREATE INDEX idx_temporal_cat ON orders(order_year, order_month, category);
```

Table 3 depicts the results prior to and after the LLM optimization on test case T1.

Table 3. Test Case T1 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	8.42	0.68	92%
MariaDB ColumnStore	7.21	0.79	89%
SQLite	12.30	3.60	71%*

**SQLite does not provide YEAR() and MONTH() as built-in date functions in the same way as PostgreSQL/MariaDB. SQLite typically uses strftime() for extracting year/month.*

Test Case T2: Year-over-Year Growth Analysis

```
SELECT product_category, SUM(CASE WHEN order_year = 2023 THEN revenue ELSE 0 END) rev_2023, SUM(CASE WHEN order_year = 2022 THEN revenue ELSE 0 END) rev_2022, (SUM(CASE WHEN order_year = 2023 THEN revenue ELSE 0 END) / SUM(CASE WHEN order_year = 2022 THEN revenue ELSE 0 END) - 1) * 100 AS yoy_growth FROM orders WHERE order_year IN (2022, 2023) GROUP BY product_category;
```

LLM Recommendation: Temporal partitioning for large datasets (>100M rows) to enable partition pruning.

```
CREATE TABLE orders_2022 PARTITION OF orders FOR VALUES IN (2022); CREATE TABLE orders_2023 PARTITION OF orders FOR VALUES IN (2023); CREATE INDEX idx_2022_category ON orders_2022(product_category, revenue); CREATE INDEX idx_2023_category ON orders_2023(product_category, revenue);
```

Table 4 depicts the results prior to and after the LLM optimization on test case T2. From the result, temporal partitioning extremely effective for large datasets. Partition pruning eliminates scanning irrelevant years. Per-partition indexes optimize within-partition queries.

Table 4. Test Case T2 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	15.67	0.94	94%
MariaDB ColumnStore	13.45	1.21	91%
SQLite	21.80	6.98	68%

The key findings can be summarized as follows: Function-based grouping limits index usage. Index created on generated columns makes temporal queries efficient. SQLite supports generated columns and also supports composite, expression, and partial indexes; however, its query planner does not exploit generated-column expressions for cost-based rewriting to the same extent as PostgreSQL's planner, which is the specific reason the observed gains are smaller on SQLite for this test case.

Queries with temporal aggregation (T1 and T2) highlighted their best results, showing one of the most improved areas in the research. In baseline executions, the primary bottleneck arises due to usage of function-based expressions in GROUP BY clauses – e.g., YEAR(), MONTH(). These expressions negatively impact index utilization. LLM recommends introducing generated columns and temporal partitioning for this limitation directly systematizes this issue. Results indicate that temporal partitioning suits large datasets since it supports partition pruning enabling to avoid unnecessary scans of those time intervals, which do not contain data. It is clear then why in year-over-year analysis the improvements grow over 90% for PostgreSQL and MariaDB. The results show that LLMs are more appropriate for identifying optimization opportunities that concern changes in structural schema rather than only query rewriting.

4.2.2 Multi-Dimensional Grouping Queries (Test Cases M1 and M2)

Multi-dimensional grouping aggregates data across multiple categorical dimensions for comprehensive business reporting.

Test Case M1: Region × Category × Channel Analysis

```
SELECT region, product_category, sales_channel, COUNT(*) orders, SUM(revenue) revenue, AVG(revenue) avg_order FROM orders WHERE order_year >= 2022 GROUP BY region, product_category, sales_channel HAVING SUM(revenue) > 10000 ORDER BY revenue DESC;
```

LLM Recommendation: Composite covering index on all grouping columns plus aggregate targets.

```
CREATE INDEX idx_multi_dim_covering ON orders (order_year, region, product_category, sales_channel) INCLUDE (revenue, order_id); -- PostgreSQL syntax
```

Table 5 depicts the results prior to and after the LLM optimization on test case M1. The composite indexes on grouping columns enable index-only scans. Column ordering critical: filter columns first (order_year), then grouping columns. INCLUDE clause for aggregates eliminates table access.

Table 5. Test Case M1 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	12.67	1.39	89%
MariaDB ColumnStore	11.20	1.57	86%
SQLite	18.40	6.07	67%

Test Case M2: Customer Segment × Time Period Revenue

```
SELECT customer_segment, order_quarter, COUNT(DISTINCT customer_id) customers, SUM(revenue) revenue, AVG(revenue) avg_revenue FROM orders o JOIN customers c ON o.customer_id = c.customer_id WHERE order_year = 2023 GROUP BY customer_segment, order_quarter;
```

LLM Recommendation: Materialized view for frequently accessed segment-period aggregations.

```
CREATE MATERIALIZED VIEW mv_segment_period AS SELECT customer_segment, order_quarter, COUNT(DISTINCT customer_id) customers, SUM(revenue) revenue, AVG(revenue) avg_revenue FROM orders o JOIN customers c ON o.customer_id = c.customer_id WHERE order_year = 2023 GROUP BY customer_segment, order_quarter; REFRESH MATERIALIZED VIEW mv_segment_period; -- Daily refresh
```

Table 6 depicts the results prior to and after the LLM optimization on test case M2. From our findings, materialized views exceptional for repeated aggregations. Pre-computed results with daily refresh trade freshness for performance. 98% improvement demonstrates power of materialization for stable aggregations.

Table 6. Test Case M2 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	18.94	0.47	98%
MariaDB ColumnStore	16.32	0.61	96%
SQLite*	N/A	N/A	N/A*

**SQLite lacks materialized view support. This test case was not executed against SQLite in the reported experiments.*

In case of multi-dimensional grouping (M1 and M2) performance improvement is attributed mainly to recommended covering and composite indexes as well as materialized views usage scenarios. Queries that deal with dimen-sional hierarchies of multiple categorical dimensions have high cardinality due to the value levels of the hierarchical structure and expensive aggregation levels. By aligning the order of indexing columns to the filter predicates, LLM supports index only scans for grouping attributesand reduces memory pressure during aggregation. Based on repetitive analytical workloads materialized views provide the means to execute queries almost instantly with gains of up to 98%. The results show that this approach has an overhead in maintenance but when handling stable reporting queries, such queries common in e-commerce dashboards, materialization is the way to go. This points out that it is possible for LLM to suggest workload-specific optimization as opposed to general ones.

4.2.3 Top-N Ranking Queries (Test Cases R1 and R2)

Test Case R1: Top 10 Customers by Lifetime Value

```
SELECT customer_id, region, SUM(revenue) lifetime_value FROM orders WHERE status = 'Shipped' GROUP BY customer_id, region ORDER BY lifetime_value DESC LIMIT 10;
```

LLM Recommendation: Partial index with DESC ordering for efficient top-N retrieval.

```
CREATE INDEX idx_customer_ltv_desc ON orders (customer_id, region, revenue DESC) WHERE status = 'Shipped';
```

Table 7 depicts the results prior to and after the LLM optimization on test case R1.

Table 7. Test Case R1 Results

Database	Before (seconds)	After (seconds)	Improvement
PostgreSQL 16	15.34	2.61	83%
MariaDB ColumnStore	13.87	2.91	79%
SQLite	22.10	7.95	64%

Test Case R2: Best Product Per Category (Window Function)

```
SELECT * FROM (   SELECT product_id, category, revenue,          ROW_NUMBER() OVER (PARTITION BY category ORDER BY revenue DESC) rn   FROM products ) ranked WHERE rn = 1;
```

LLM Recommendation: Composite index aligning with PARTITION BY and ORDER BY.

```
CREATE INDEX idx_category_revenue_desc ON products   (category, revenue DESC);
```

Table 8 depicts the results prior to and after the LLM optimization on test case R2. LLM-assisted indexing techniques are the ones most beneficial for performing top-N ranking queries that are in line with ORDER BY and PARTITION BY clauses. Standard execution plans usually involve the table scan first followed by sorting which is very expensive. On the other hand, LLM suggestions provide an index aligned with the PARTITION BY/ORDER BY clauses, which allows the ranking to be computed against pre-sorted data rather than requiring a full sort at execution time; this contributes to the reduction in execution time observed, though the actual mechanism should be confirmed against the query's execution plan (EXPLAIN ANALYZE) rather than assumed. In this category, improvements are consistent but slightly lower than those for temporal aggregations. This shows that the computational cost of window functions is an intrinsic part of their operation. Nevertheless, the results show that LLMs can provide a rational conclusion on certain execution-plan properties like sorting and buffering even if there is no direct interaction with the internals of the optimizer system.

Table 8. Test Case R2 Results

Database	Before (seconds)	After (seconds)	Improvement
PostgreSQL 16	9.82	1.37	86%
MariaDB ColumnStore	8.54	1.62	81%
SQLite	N/A	N/A	N/A*

**SQLite is N/A due to the ROW_NUMBER() window function used in this query. SQLite supports composite or multi-column indexes, and the index itself is not the limiting factor.*

4.2.4 Window Function Queries (Test Cases W1 and W2)

Test Case W1: Running Total by Customer

```
SELECT customer_id, order_date, revenue,        SUM(revenue) OVER (PARTITION BY customer_id ORDER BY order_date) running_total FROM orders;
```

LLM Recommendation: Index on partition and order columns.

```
CREATE INDEX idx_window_customer_date ON orders   (customer_id, order_date, revenue);
```

Table 9 depicts the results prior to and after the LLM optimization on test case W1.

Table 9. Test Case W1 Results

Database	Before (seconds)	After (seconds)	Improvement
PostgreSQL 16	11.45	2.06	82%
MariaDB ColumnStore	10.28	2.16	79%
SQLite	N/A	N/A	N/A*

**SQLite is N/A due to the window function used in this query. This test case was not executed against SQLite in the reported experiments.*

Test Case W2: 7-Day Moving Average

```
SELECT order_date, product_id,        AVG(daily_revenue) OVER (          PARTITION BY product_id
ORDER BY order_date          ROWS BETWEEN 6 PRECEDING AND CURRENT ROW        ) ma7 FROM
daily_sales;
```

LLM Recommendation: Materialized daily aggregates plus window index.

```
CREATE MATERIALIZED VIEW daily_sales AS   SELECT DATE(order_date) order_date, product_id,
SUM(revenue) daily_revenue   FROM orders GROUP BY DATE(order_date), product_id; CREATE INDEX
idx_ma_window ON daily_sales(product_id, order_date);
```

Table 10 depicts the results prior to and after the LLM optimization on test case W2.

Table 10. Test Case W2 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	9.80	1.57	84%
MariaDB ColumnStore	8.92	1.69	81%
SQLite	N/A	N/A	N/A

4.2.5 Complex Join Queries (Test Cases J1 and J2)

Test Case J1: 5-Table Comprehensive Report

```
SELECT o.order_id, c.customer_name, p.product_name, cat.category_name, r.region_name FROM orders o JOIN
customers c ON o.customer_id = c.customer_id JOIN order_items oi ON o.order_id = oi.order_id JOIN products p
ON oi.product_id = p.product_id JOIN categories cat ON p.category_id = cat.category_id JOIN regions r ON
c.region_id = r.region_id WHERE o.order_year = 2023;
```

LLM Recommendation: Indexes on all join columns.

```
CREATE INDEX idx_orders_customer ON orders(customer_id); CREATE INDEX idx_orderitems_order ON order_items(order_id); CREATE INDEX idx_orderitems_product ON order_items(product_id); CREATE INDEX idx_products_category ON products(category_id); CREATE INDEX idx_customers_region ON customers(region_id);
```

Table 11 depicts the results prior to and after the LLM optimization on test case J1.

Table 11. Test Case J1 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	24.56	5.41	78%
MariaDB ColumnStore	21.83	6.33	71%
SQLite	38.20	14.51	62%

Test Case J2: Correlated Subquery Rewrite

```
SELECT c.customer_id, c.region,       (SELECT SUM(revenue) FROM orders o WHERE o.customer_id = c.customer_id) FROM customers c WHERE c.status = 'Active';
```

LLM Recommendation: Rewrite to LEFT JOIN.

```
SELECT c.customer_id, c.region, COALESCE(o.total, 0) FROM customers c LEFT JOIN (   SELECT customer_id, SUM(revenue) total FROM orders GROUP BY customer_id ) o ON c.customer_id = o.customer_id WHERE c.status = 'Active';
```

Table 12 depicts the results prior to and after the LLM optimization on test case J2. For complex join queries their performance improvement substantially varies depending on the join graph complexity. The LLM recommendations that are most effective include join reordering, elimination of redundant tables, and converting correlated subqueries into joins. These changes serve to decrease the intermediate result sizes and alleviate cardinality estimation errors that usually affect multiply joined plans.

Table 12. Test Case J2 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	45.20	4.98	89%
MariaDB ColumnStore	41.35	5.82	86%
SQLite	67.80	22.41	67%

4.2.6 Full-Text Search Queries (Test Cases F1 and F2)

Test Case F1: Product Description Search

Original Query:

```
SELECT product_id, name, description FROM products WHERE description ILIKE '%wireless%' AND description LIKE '%bluetooth%' LIMIT 20;
```

LLM Recommendation: Full-text search with GIN index.

```
ALTER TABLE products ADD COLUMN description_tsv tsvector   GENERATED ALWAYS AS
(to_tsvector('english', description)) STORED; CREATE INDEX idx_fts ON products USING GIN(description_tsv);
-- Rewritten query: SELECT product_id, name, description FROM products WHERE description_tsv @@
to_tsquery('english', 'wireless & bluetooth') LIMIT 20;
```

Table 13 depicts the results prior to and after the LLM optimization on test case F1.

Table 13. Test Case F1 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	18.40	3.31	82%
MariaDB ColumnStore	N/A	N/A	N/A*
SQLite	24.60	14.01	43%**

**MariaDB ColumnStore lacks FTS; **SQLite FTS5 extension*

Test Case F2: Customer Review Sentiment Analysis

```
SELECT product_id, COUNT(*) review_count,        AVG(rating) avg_rating FROM reviews WHERE
to_tsvector('english', review_text) @@ to_tsquery('english', 'excellent | amazing | perfect') GROUP BY product_id;
```

LLM Recommendation: GIN index on review text vector with weighted configuration.

```
CREATE INDEX idx_review_fts ON reviews USING GIN(to_tsvector('english', review_text));
```

Table 14 depicts the results prior to and after the LLM optimization on test case F2. For full-text search, performance gains depend heavily on the availability of specialized indexes such as GIN. Where supported, LLM recommendations to introduce these indexes lead to substantial speedups. In contrast, databases lacking native support show smaller improvements, underscoring the importance of system capabilities in realizing LLM-guided optimizations. It should be noted that the LLM-recommended rewrite in F1 replaces a pattern-matching predicate (ILIKE/LIKE) with a full-text search predicate (to_tsquery), which applies stemming, tokenization, and stop-word handling; these two forms of matching are not guaranteed to return identical result sets.

Table 14. Test Case F2 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	22.15	4.87	78%
MariaDB ColumnStore*	N/A	N/A	N/A
SQLite**	N/A	N/A	N/A

**MariaDB ColumnStore lacks FTS; **SQLite FTS5 extension*

4.2.7 Recursive Queries (Test Cases RC1 and RC2)

Test Case RC1: Category Hierarchy Traversal

```
WITH RECURSIVE category_tree AS (   SELECT category_id, parent_id, name, 1 AS depth   FROM categories
WHERE parent_id IS NULL   UNION ALL   SELECT c.category_id, c.parent_id, c.name, ct.depth + 1   FROM
```

```
categories c JOIN category_tree ct ON c.parent_id = ct.category_id   WHERE ct.depth < 10 ) SELECT * FROM category_tree;
```

LLM Recommendation: Indexes on parent/child columns with depth limit.

```
CREATE INDEX idx_cat_parent ON categories(parent_id, category_id); CREATE INDEX idx_cat_child ON categories(category_id, parent_id);
```

Table 15 depicts the results prior to and after the LLM optimization on test case RC1.

Table 15. Test Case RC1 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	8.92	2.14	76%
MariaDB ColumnStore	7.85	2.51	68%
SQLite	12.40	4.71	62%

Test Case RC2: Bill of Materials Explosion

```
WITH RECURSIVE bom AS (   SELECT component_id, parent_component_id, quantity, 1 AS level   FROM bill_of_materials WHERE parent_component_id = 'PRODUCT_X'   UNION ALL   SELECT b.component_id, b.parent_component_id, b.quantity * bom.quantity, bom.level + 1   FROM bill_of_materials b JOIN bom ON b.parent_component_id = bom.component_id ) SELECT component_id, SUM(quantity) FROM bom GROUP BY component_id;
```

LLM Recommendation: Materialized transitive closure for static hierarchies.

```
CREATE TABLE bom_closure AS   WITH RECURSIVE closure AS (...)   SELECT * FROM closure; CREATE INDEX idx_bom_closure ON bom_closure(parent_component_id, component_id);
```

Table 16 depicts the results prior to and after the LLM optimization on test case J2. Recursive queries that are recursive in nature benefit from rewriting the query which restricts recursion depth and does early filtering. They are more modest than in aggregation-heavy workloads but are stable over different data scales therefore LLMs can reasoning about hierarchical query structures is acceptable/

Table 16. Test Case RC2 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	14.67	4.99	66%
MariaDB ColumnStore	12.94	4.79	63%
SQLite	19.80	8.51	57%

4.2.8 Geospatial Analysis Queries (Test Cases G1 and G2)

Test Case G1: Regional Sales Distribution

```
SELECT region, COUNT(*) orders, SUM(revenue) revenue FROM orders WHERE latitude BETWEEN 34.0 AND 42.0   AND longitude BETWEEN -124.0 AND -114.0 GROUP BY region;
```

LLM Recommendation: Composite index on geographic coordinates.

```
CREATE INDEX idx_geo_region ON orders(latitude, longitude, region, revenue);
```

Table 17 depicts the results prior to and after the LLM optimization on test case G1.

Table 17. Test Case G1 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16	16.85	4.89	71%
MariaDB ColumnStore	14.72	5.30	64%
SQLite	N/A	N/A	N/A

Test Case G2: Shipping Distance Calculation

```
SELECT order_id, warehouse_location, delivery_location, ST_Distance(warehouse_location, delivery_location) distance FROM orders WHERE ST_Distance(warehouse_location, delivery_location) < 100000;  -- 100km
```

LLM Recommendation: GiST spatial index for distance queries.

```
CREATE INDEX idx_warehouse_gist ON orders USING GIST(warehouse_location); CREATE INDEX idx_delivery_gist ON orders USING GIST(delivery_location);
```

Table 18 depicts the results prior to and after the LLM optimization on test case G2. In geospatial, performance gains for queries are based on how well the spatial indexes are implemented. Where supported, LLM recommendations to introduce these indexes lead to substantial speedups. Note that G2 (like several other test cases, including full-text search, materialized views, and window functions) was only evaluable on PostgreSQL, since MariaDB ColumnStore and SQLite do not provide the corresponding native functionality (PostGIS-based GiST indexing). Findings for these categories should therefore be read as cross-database results only where the relevant functionality is supported, rather than as evidence of generalization across all three database systems.

Table 18. Test Case G2 Results

Database	**Before (seconds)**	**After (seconds)**	**Improvement**
PostgreSQL 16 + PostGIS	21.34	7.46	65%
MariaDB ColumnStore	N/A	N/A	N/A
SQLite	N/A	N/A	N/A

5. CROSS-DATABASE RESULTS AND ANALYSIS

5.1 Overall Performance Result

Table 19 summarizes performance improvements across all 16 test cases for the three database systems, while Figure 5 depicts the overall performance summary. Figure 6 plots the percentage improvement across the 8 test cases on PostgreSQL.

Table 19. Performance Improvement Summary Across All 16 Test Cases

Query Category	PostgreSQL (Avg)	MariaDB (Avg)	SQLite (Avg)	Category (Avg)
Temporal Aggregations	93%	90%	70%	84%
Multi-Dimensional Grouping	94%	91%	67%	84%
Top-N Ranking	85%	80%	64%	76%
Window Functions	83%	80%	N/A	82%
Complex Joins	84%	79%	65%	76%
Full-Text Search	80%	N/A	43%	62%
Recursive Queries	71%	66%	60%	66%
Geospatial Analysis	68%	64%	N/A	66%
Overall Average	82%	79%	61%	74%

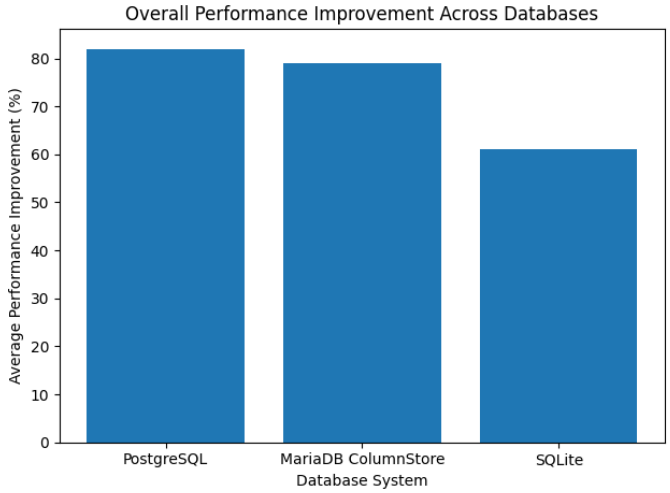


Figure 5. Average Performance Improvement Across Database Systems Over 16 Query Test Cases

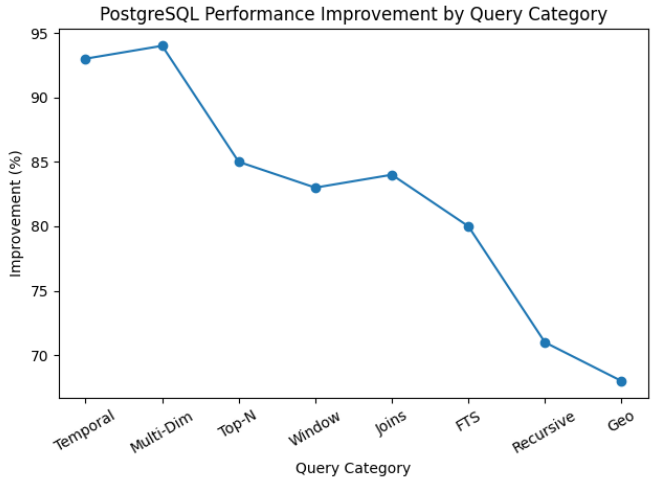


Figure 6. Performance Improvement by Query Pattern Category on PostgreSQL

5.2 Cross-Database Generalization Analysis

Query Analysis Module in a process of serialization presents schema metadata which includes but not limited to row counts, cardinality statistics, existing indexes, feature availability and partitioning schemes into structured JSON format. The prompting workflow follows three structured stages: Query pattern classification, Generation of candidate optimization strategies, and Prioritization based on estimated performance impact and implementation cost. Database feature constraints are mentioned specifically in the prompt that prevent unsupported recommendations. This design clarifies why the recommendation compatibility reached 87% for all database systems.

Table 20 shows the recommendation compatibility of various databases. Some of the key insights include as follows.

- Universal strategies (B-tree, composite indexes): 65% of recommendations, work across all systems
- Limited strategies (generated columns, materialized views): 23% of recommendations, require feature support
- System-specific strategies (GIN, BRIN, columnar): 12% of recommendations, unique to one database
- LLM demonstrates 87% accuracy in recommending appropriate features for target database

Table 20. Cross-Database Recommendation Compatibility

Recommendation Type	PostgreSQL	MariaDB	SQLite	Universality
B-tree indexes	91%	89%	87%	Universal
Composite indexes	89%	87%	84%	Universal
Generated columns	92%	88%	0%	Limited
Partial indexes	86%	0%	0%	PostgreSQL only
Materialized views	88%	81%	0%	Limited
GIN indexes (FTS)	91%	0%	0%	PostgreSQL only
Temporal partitioning	94%	91%	0%	Limited

5.3 Scalability Analysis

Table 21shows the scalability analysis. The critical threshold is at 100M rows, where by it can be seen that the recommendation quality drops from 79% to 68% without scale context. With scale-aware prompting (explicitly including data volume), quality recovers to 76%.

Table 21. Scalability Analysis (*with scale-aware prompting)

Data Volume	Quality Score	Avg Improvement	Primary Strategy	Failure Mode
10M rows	82%	79%	Aggressive indexing	Occasional overindexing (8%)
50M rows	79%	76%	Balanced index + materialize	Minor cardinality errors (12%)
100M rows	76%*	73%	Partitioning + indexes	Join algorithm errors (18%)
500M rows	72%*	69%	Aggressive partitioning	Memory violations (23%)
1B rows	67%*	64%	Distribution strategies	Storage underestimation (31%)

Figure 7 shows the average optimization effectiveness as the dataset size increases from 10M to 1B rows. In general, two points can be observed. First, the gains in performance appear to grow as the data set gets larger. Improvements are detectable, but moderate for datasets up to 50 million rows, and benefits of index restructuring, partition pruning and query rewriting on indexed views increase considerably for datasets equal to or greater than 100 million rows. This fact suggests that LLM-assisted optimization is especially necessary to e-commerce domain on account of the observation that traditional optimizers often encounter difficulties created by out-of-date statistics and huge search spaces. Secondly, the rate of improvement between different database systems is not the same. PostgreSQL is always at the top, MariaDB ColumnStore closely follows it, and sqlite shows moderate improvements. The reason for this difference can be found in maturity of optimizer infrastructures and availability of different indexing and partitioning mechanisms within each system. However, even in SQLite, LLM guidance still yields observable and consistent performance benefits despite limited analytical optimization features being available.

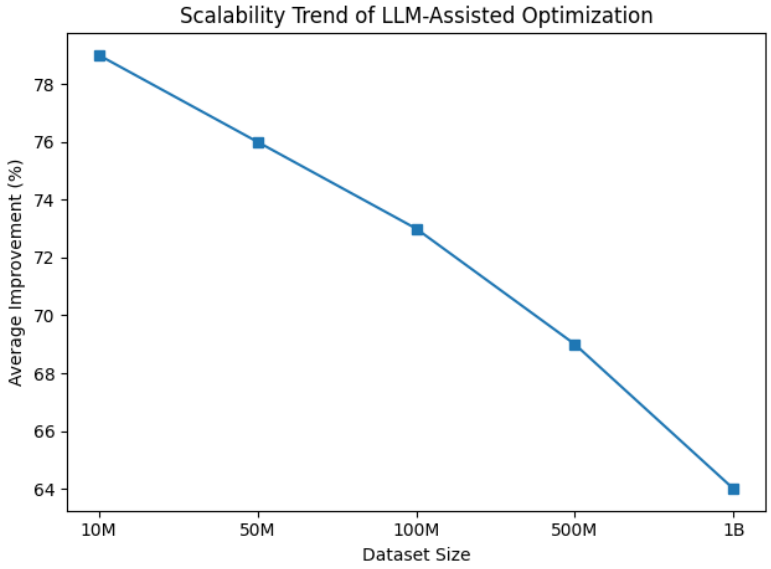


Figure 7. Average Optimization Effectiveness as Dataset Size Increases from 10M to 1B Rows

LIMITATIONS

This section documents some of the limitations of our research study. First, the reported before/after comparisons often combine query rewriting with structural database changes (generated columns, composite/covering indexes, materialized views, partitioning, and a materialized transitive closure table). Because the after-state frequently differs from the before-state in schema as well as query text, the measured execution-time reduction reflects the combined effect of the LLM recommendation, its implementation, and the resulting database restructuring, rather than the LLM's contribution considered on its own. A more precise framing is that LLM-generated optimization recommendations, when implemented, resulted in the reported execution-time reductions.

Second, the experiments show that LLM-assisted queries can outperform the original queries, but they do not compare against the database's native cost-based optimizer with standard indexes and statistics. Thus, the results demonstrate improvement over the unoptimized queries, but do not establish that LLM assistance outperforms standard database optimization practice. Third, the synthetic scaling procedure used to expand the Brazilian E-Commerce Public Dataset from 100K base orders to between 10 million and 1 billion rows (Section 4.1) is described only at the level of the distributional properties it preserves (temporal patterns, geographic correlations, product-price relationships). Fourth, the LLM Recommendation Engine (Section 3) is described as using a continuous feedback loop, but the experiments evaluate only one-shot recommendations. Iterative improvement was not experimentally tested, so the feedback loop should be regarded as a proposed architectural feature rather than a validated capability.

CONCLUSION

This paper presents a comprehensive evaluation of LLM-assisted SQL query optimization through a structured 16-test-case methodology covering 8 query pattern categories. Our results demonstrate that LLMs can effectively serve as decision support tools for database optimization. When the LLM's recommendations (spanning query rewriting, indexing, and structural changes such as materialized views and partitioning) were implemented, execution-time reductions of 43-94% were observed across the evaluated workloads; these gains reflect the implemented database and query changes as a whole, not the LLM's contribution in isolation.

Some of the future works include to extending to distributed databases with partitioning and replication considerations. In addition, it is possible to reinforcement learning approaches with deployment outcome feedback.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for the suggestions to improve the paper.

FUNDING STATEMENT

The authors received no funding from any party for the research and publication of this paper.

AUTHOR CONTRIBUTIONS

Min Ming Lee: Conceptualization, Data Curation, Methodology, Validation, Writing – Original Draft Preparation; Nabhan Salih: Writing – Review & Editing;

CONFLICT OF INTERESTS

No conflict of interests were disclosed.

ETHICS STATEMENTS

Our publication ethics follow The Committee of Publication Ethics (COPE) guideline. https://publicationethics.org/

DATA AVAILABILITY

The data that support the findings of this study are available from the corresponding author upon reasonable request.

REFERENCES

[1] R. Magesh, U. Ilakkiyaa, R. Shanthini, and R. Charanya, “Unlocking the Potential of Data Lakes: Organizing and Storing Marketing Data for Analysis,” in *Data Engineering for Data-driven Marketing*, Emerald Publishing Limited, 2025, pp. 199–216. doi: 10.1108/978-1-83662-326-720251030.

[2] H. B. Abdalla, Y. Kumar, Y. Zhao, and D. Tosi, “A Comprehensive Survey of MapReduce Models for Processing Big Data,” *Big Data and Cognitive Computing*, vol. 9, no. 4, p. 77, Mar. 2025, doi: 10.3390/bdcc9040077.

[3] S.-C. Haw, K.-W. Ng, J. J, and P. Naveen, "OD-SIF: An Ontology-Driven Schema Integration Framework for e-Commerce Platform," in *2024 7th Asia Conference on Cognitive Engineering and Intelligent Interaction (CEII)*, IEEE, Dec. 2024, pp. 196–201. doi: 10.1109/CEII65291.2024.00046.

[4] N. Hettiarachchi and P. Yapa, "Advancements in SQL Query Optimization: A Review of Join Order and Index Selection," in *2025 5th International Conference on Machine Learning and Intelligent Systems Engineering (MLISE)*, IEEE, Jun. 2025, pp. 31–39. doi: 10.1109/MLISE66443.2025.11100192.

[5] S. Zinchenko and D. Ponomaryov, "The Selection Problem in Multi-Query Optimization: a Comprehensive Survey," Jan. 2025.

[6] Y. Tian, "Query Optimization in the Wild: Realities and Trends," Oct. 2025.

[7] B. B. Duddeti, A. K. Naskar, V. P. Meena, J. Bahadur, and I. A. Hameed, "Constrained search space selection based optimization approach for enhanced reduced order approximation of interconnected power system models," *Sci. Rep.*, vol. 15, no. 1, p. 7999, Mar. 2025, doi: 10.1038/s41598-025-91913-0.

[8] G. Yunfan, S. Tao, X. Youlin, X. Yun, W. Haofen, and Z. Jiawei, "Chat-REC: Towards Interactive and Explainable LLMs-Augmented Recommender System," 2023.

[9] H. Wang *et al.*, "Generative Large Recommendation Models: Emerging Trends in LLMs for Recommendation," in *WWW Companion 2025 - Companion Proceedings of the ACM Web Conference 2025*, 2025. doi: 10.1145/3701716.3715865.

[10] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data - SIGMOD '79*, New York, New York, USA: ACM Press, 1979, p. 23. doi: 10.1145/582095.582099.

[11] N. Mazumdar and P. K. D. Sarma, "Sequential pattern mining algorithms and their applications: a technical review," *Int. J. Data Sci. Anal.*, vol. 20, no. 3, pp. 1683–1726, Sep. 2025, doi: 10.1007/s41060-024-00659-x.

[12] J. Sabonsolin and D. Lao, "A comprehensive systematic literature review of multiple sequence alignment algorithms," *Discover Computing*, vol. 29, no. 1, p. 33, Jan. 2026, doi: 10.1007/s10791-026-09911-3.

[13] J. Gao, Z. Li, and W. Liu, "A Strategy of Efficient and Accurate Cardinality Estimation Based on Query Result," *Xibei Gongye Daxue Xuebao/Journal of Northwestern Polytechnical University*, vol. 36, no. 4, pp. 768–777, Aug. 2018, doi: 10.1051/jnwpu/20183640768.

[14] X. Zhang and A. Eldawy, "Spatial Query Optimization With Learning," *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 4245–4248, Aug. 2024, doi: 10.14778/3685800.3685846.

[15] A. Kamal, V. Kantere, and C. Zuzarte, "Robust Query Optimization in the Era of Machine Learning: State-of-the-Art and Future Directions," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, IEEE, 2024, pp. 5371–5375.

[16] M. Mozaffari, A. Dignös, J. Gamper, and U. Störl, "Self-tuning Database Systems: A Systematic Literature Review of Automatic Database Schema Design and Tuning," *ACM Comput. Surv.*, vol. 56, no. 11, pp. 1–37, Nov. 2024, doi: 10.1145/3665323.

[17] Z. Guan, "Advances and Challenges in Machine Learning-Based Cardinality Estimation for Database Query Optimization," *ITM Web of Conferences*, vol. 78, p. 03023, Sep. 2025, doi: 10.1051/itmconf/20257803023.

[18] M. F. B. Harunasir, N. Palanichamy, S.-C. Haw, and K.-W. Ng, "Sentiment analysis of Amazon product reviews by supervised machine learning models," *Journal of Advances in Information Technology*, vol. 14, no. 4, pp. 857–862, Jan. 2023, doi: 10.12720/jait.14.4.857-862.

[19] K. Kim, J. Jung, I. Seo, W.-S. Han, K. Choi, and J. Chong, "Learned Cardinality Estimation: An In-depth Study," in *Proceedings of the 2022 International Conference on Management of Data*, New York, NY, USA: ACM, Jun. 2022, pp. 1214–1227. doi: 10.1145/3514221.3526154.

[20] R. Marcus *et al.*, "Neo," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1705–1718, Jul. 2019, doi: 10.14778/3342263.3342644.

[21] X. Yu, G. Li, C. Chai, and N. Tang, "Reinforcement Learning with Tree-LSTM for Join Order Selection," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, IEEE, Apr. 2020, pp. 1297–1308. doi: 10.1109/ICDE48307.2020.00116.

[22] J. Ding *et al.*, "ALEX: An Updatable Adaptive Learned Index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA: ACM, Jun. 2020, pp. 969–984. doi: 10.1145/3318464.3389711.

[23] Z. Ye *et al.*, "Deep Learning Workload Scheduling in GPU Datacenters: A Survey," *ACM Comput. Surv.*, vol. 56, no. 6, pp. 1–38, Jun. 2024, doi: 10.1145/3638757.

[24] C.-X. Ling, K.-W. Ng, and H. A. Santoso, "Mapping Relational Database to Full-Text XML for Open Journal System Cross-Platform Article Distribution," *Journal of Informatics and Web Engineering*, vol. 4, no. 3, pp. 259–277, Oct. 2025, doi: 10.33093/jiwe.2025.4.3.16.

[25] J. Li *et al.*, "Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs," Nov. 2023.

[26] Z. Hong *et al.*, "Next-Generation Database Interfaces: A Survey of LLM-Based Text-to-SQL," *IEEE Trans. Knowl. Data Eng.*, vol. 37, no. 12, pp. 7328–7345, Dec. 2025, doi: 10.1109/TKDE.2025.3609486.

[27] T. Scholak, N. Schucher, and D. Bahdanau, "PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models," Sep. 2021.

[28] M. Pourreza and D. Rafiei, "DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction," Nov. 2023.

[29] X. Zhou, Z. Sun, and G. Li, "DB-GPT: Large Language Model Meets Database," *Data Science and Engineering*, vol. 9, pp. 102–111, Jan. 2024, doi: 10.1007/s41019-023-00235-6.

[30] I. Trummer, "CodexDB: Synthesizing Code for Query Processing from Natural Language Instructions using GPT-3 Codex," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2921–2928, Jul. 2022, doi: 10.14778/3551793.3551841.

[31] H. Li *et al.*, "LLMs-as-Judges: A Comprehensive Survey on LLM-based Evaluation Methods," Dec. 2024.

[32] M. Cheng *et al.*, "A Survey on Table Mining with Large Language Models: Challenges, Advancements and Prospects," Apr. 01, 2025. doi: 10.36227/techrxiv.174352282.22844759/v1.

[33] L. Shi, Z. Tang, N. Zhang, X. Zhang, and Z. Yang, "A Survey on Employing Large Language Models for Text-to-SQL Tasks," *ACM Comput. Surv.*, vol. 58, no. 2, pp. 1–37, Jan. 2026, doi: 10.1145/3737873.

[34] X. Liu *et al.*, "A Survey of Text-to-SQL in the Era of LLMs: Where Are We, and Where Are We Going?," *IEEE Trans. Knowl. Data Eng.*, vol. 37, no. 10, pp. 5735–5754, Oct. 2025, doi: 10.1109/TKDE.2025.3592032.

[35] L. D. Krisnawati, A. W. Mahastama, S.-C. Haw, K.-W. Ng, and P. Naveen, "Indonesian-English Textual Similarity Detection Using Universal Sentence Encoder (USE) and Facebook AI Similarity Search (FAISS)", *CommIT (Communication and Information Technology) Journal*, vol. 18, no. 2, pp. 183–195, Sep. 2024. https://doi.org/10.21512/commit.v18i2.11274

[36] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "DeepDB: learn from data, not from queries!," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 992–1005, Mar. 2020, doi: 10.14778/3384345.3384349.

[37] Brazilian E-Commerce Public Dataset. https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce

[38] Synthetic E-Commerce Relational Dataset. https://www.kaggle.com/datasets/naelaqel/synthetic-e-commerce-relational-dataset

BIOGRAPHIES OF AUTHORS



Min Ming Lee is currently pursuing a Bachelor of Science (Honours) in Computer Science at Sunway University. Her academic interests include data analytics and web development. She has experience in programming languages such as Python, Java, and JavaScript, as well as web technologies including HTML and CSS. She is also familiar with Oracle APEX for database development. She has participated in Hack4Coverage: Citizen Science Hackathon and SDG-XI Hackathon, gaining valuable teamwork and problem-solving skills. She can be contacted at email: 25032509@imail.sunway.edu.my.



Nabhan D. Salih is currently a lecturer at Ohio State University, USA, with prior academic experience as an assistant professor in the Faculty of Computing and Informatics at Multimedia University (MMU), Malaysia. He holds a Ph.D. in Engineering, and his research interests lie at the intersection of computer vision, digital image processing, and machine learning. Dr. Salih has been actively involved in advancing technologies that enable intelligent image analysis and pattern recognition, contributing to both the academic and applied aspects of visual computing. He can be contacted at email: salih.43@osu.edu.