E-ISSN 0000-0000
Applied Science: Jurnal Sains dan Teknologi Terapan
Vol 1, No 1 (2026)
Website: https://journal.emergingmdtrends.com/index.php/applied-science

Query Performance Optimization in MySQL Databases: An Experimental Analysis of Normalization and Indexing

Risman Kurniawan[1*], Sahal Husmah[2], Darmawan Yudhanegara[3]

[123]Digitech University, Bandung, Indonesia

*Correspondent email: risman20125003@digitechunuversity.ac.id

Received: 12 Januari 2026 **Revised:** 10 Maret 2026 **Accepted:** 14 April 2026 **Published:** 26 Juni 2026

Abstract. The rapid growth of information systems has increased the demand for database architectures that are able to provide fast, accurate, consistent, and efficient data access. However, many small- and medium-scale information systems still encounter performance problems caused by suboptimal database design, including data redundancy, update anomalies, inconsistent records, and slow query execution. These problems become more critical when the volume of stored data increases and database transactions involve frequent retrieval, insertion, and updating operations. This study aims to analyze the effect of normalization and indexing techniques on database performance optimization in a relational database environment. An experimental method was applied by comparing query execution performance before and after optimization. The experiment was conducted using MySQL as the database management system with a dataset consisting of 50,000 records. Several query operations were tested, including SELECT, INSERT, and UPDATE, to evaluate the impact of structural design improvement and index implementation on query response time. The results indicate that normalization improves database structure by reducing redundancy, minimizing data anomalies, and supporting better data consistency. In addition, indexing substantially improves retrieval performance, particularly in SELECT operations, with query execution time reduced by up to 70% after optimization. Nevertheless, the implementation of indexes must be carefully planned because excessive or inappropriate indexing may increase storage requirements and affect write operations. This study contributes practical insights for database designers, developers, and system administrators in optimizing relational database performance through appropriate schema design and indexing strategies, particularly for small- and medium-scale information systems.

Keywords: normalization; indexing; database performance; query optimization; MySQL; relational database

Abstrak. Pertumbuhan sistem informasi yang semakin cepat mendorong kebutuhan terhadap arsitektur basis data yang mampu menyediakan akses data secara cepat, akurat, konsisten, dan efisien. Namun, berbagai sistem informasi berskala kecil hingga menengah masih menghadapi permasalahan kinerja yang disebabkan oleh desain basis data yang belum optimal, seperti redundansi data, anomali pembaruan, inkonsistensi data, serta lambatnya waktu eksekusi query. Permasalahan tersebut menjadi semakin penting ketika volume data meningkat dan transaksi basis data melibatkan proses pencarian, penambahan, serta pembaruan data secara berulang. Penelitian ini bertujuan untuk menganalisis pengaruh penerapan teknik normalisasi dan indexing terhadap optimasi kinerja basis data pada

lingkungan basis data relasional. Metode penelitian yang digunakan adalah eksperimen dengan membandingkan performa eksekusi query sebelum dan sesudah dilakukan optimasi. Pengujian dilakukan menggunakan MySQL sebagai database management system dengan dataset berjumlah 50.000 record. Beberapa operasi query yang diuji meliputi SELECT, INSERT, dan UPDATE untuk menilai dampak perbaikan struktur basis data dan penerapan indeks terhadap waktu respons query. Hasil penelitian menunjukkan bahwa normalisasi mampu memperbaiki struktur basis data melalui pengurangan redundansi, penurunan potensi anomali data, dan peningkatan konsistensi data. Selain itu, indexing mampu meningkatkan performa pencarian data secara substansial, khususnya pada operasi SELECT, dengan penurunan waktu eksekusi query hingga 70% setelah optimasi. Meskipun demikian, penerapan indeks perlu dirancang secara tepat karena penggunaan indeks yang berlebihan atau tidak sesuai dapat meningkatkan kebutuhan penyimpanan dan memengaruhi operasi penulisan data. Penelitian ini memberikan kontribusi praktis bagi perancang basis data, pengembang sistem, dan administrator basis data dalam mengoptimalkan performa basis data relasional melalui desain skema dan strategi indexing yang tepat, terutama pada sistem informasi berskala kecil hingga menengah.
Kata Kunci: normalisasi; indexing; kinerja basis data; optimasi query; MySQL; basis data relasional

1. Pendahuluan

The rapid development of information technology has made databases a fundamental component of almost every modern information system. Databases are no longer used merely as data repositories, but also as critical infrastructures that support transaction processing, decision making, reporting, application integration, and real-time data access. In many organizations, the effectiveness of an information system is strongly influenced by how efficiently the database stores, retrieves, updates, and manages data. Therefore, database performance has become an important concern in the design and implementation of information systems, particularly when the volume of data continues to increase and users expect fast, accurate, and consistent access to information (Elmasri & Navathe, 2016; Gupta et al., 2024; Rahman et al., 2024).

Relational database management systems remain widely used because they provide a structured model for organizing data into tables, establishing relationships among entities, enforcing constraints, and maintaining data consistency. However, the effectiveness of a relational database does not depend only on the availability of a database management system, but also on the quality of its schema design and optimization strategy. Poorly designed databases may lead to several problems, such as data redundancy, insertion anomalies, update anomalies, deletion anomalies, inconsistent records, inefficient storage, and slow query execution. These problems can reduce the reliability of information systems and affect the speed of data processing, especially in systems that handle repeated transactions and large volumes of records (Elmasri & Navathe, 2016; Hardini et al., 2025).

One of the main techniques used to improve database structure is normalization. Normalization is a systematic process of organizing data into related tables to minimize redundancy and dependency anomalies. Through normalization, data attributes are grouped based on functional dependencies so that each table represents a clear entity or relationship. The application of normalization, especially up to the Third Normal Form (3NF), can help reduce repeated data, improve data integrity, and maintain consistency when records are inserted, updated, or deleted. Hardini et al. (2025) emphasized that normalization contributes to data storage efficiency by reducing unnecessary duplication and improving the logical structure of database design. Similarly, Firdaus et al. (2025) showed that relational database optimization through indexing and advanced normalization can improve the quality and performance of academic information systems.

Although normalization provides important benefits for data consistency and integrity, it may also increase the complexity of data retrieval. A highly normalized schema often separates data into multiple tables, which may require more join operations when executing queries. In some cases, the increase in join operations can affect query execution time if the database is not supported by proper indexing and query optimization strategies. Therefore, normalization alone is not sufficient to guarantee high database performance. It must be accompanied by an appropriate indexing strategy, query tuning, and database optimization mechanism to ensure that improved data structure does not create new performance bottlenecks (Elmasri & Navathe, 2016; Jarke & Koch, 1984; Ramakrishnan & Gehrke, 2003).

Indexing is another essential technique in database performance optimization. An index allows a database management system to locate records more efficiently without scanning all rows in a table. In relational databases, indexes are commonly applied to primary keys, foreign keys, and columns frequently used in search conditions, sorting, grouping, and join operations. The use of indexing can significantly reduce query execution time, particularly for SELECT statements that access specific rows from large tables. Gupta et al. (2024) explained that indexing is one of the essential techniques for optimizing MySQL performance, especially when a database contains a large number of records and frequently executed queries. In a similar context, Ikhwan and Pramadjaya (2026) demonstrated that database tuning through indexing and query optimization can improve the performance of an academic questionnaire system.

The theoretical foundation of query optimization has been widely discussed in database literature. Jarke and Koch (1984) described query optimization as a central issue in database systems because query execution can be performed using different access paths and execution strategies. Selinger et al. (1979) introduced the concept of access path selection in relational database systems, which became one of the foundations of cost-based query optimization. In cost-based optimization, the database optimizer evaluates alternative query execution plans and selects the plan with the lowest estimated cost based on statistics, indexes, join methods, and access paths. Chaudhuri (1998) further explained that query optimization in relational database systems is a complex process because the optimizer must consider multiple alternatives before executing a query. These foundational studies show that query performance is influenced not only by the SQL statement itself, but also by schema design, index availability, data distribution, statistics, and the optimizer's ability to select an efficient execution plan.

In MySQL databases, performance optimization is particularly relevant because MySQL is widely used in web-based applications, academic information systems, business systems, and small to medium-scale enterprise applications. MySQL provides various mechanisms for improving query performance, including indexing, query plan analysis, optimizer statistics, and execution plan inspection. However, in practice, many database implementations still experience slow query response because indexes are not applied to appropriate columns, database schemas are not normalized properly, or query structures are not designed efficiently. Rahman et al. (2024) noted that advanced SQL query optimization is increasingly important in real-time big data analytics because inefficient queries can affect system responsiveness and analytical processing. This issue becomes more relevant when databases are required to support both transactional and analytical workloads.

Several recent studies have examined database optimization using normalization and indexing. Firdaus et al. (2025) analyzed relational database optimization using indexing and advanced normalization in an academic information system and found that these techniques can improve database structure and query performance. Hardini et al. (2025) discussed the application of database normalization in increasing data storage efficiency and emphasized the role of normalization in reducing redundancy. Gupta et al. (2024) presented several essential techniques for optimizing MySQL

performance, including indexing and query improvement. Ikhwan and Pramadjaya (2026) applied database tuning through indexing and query optimization in an academic questionnaire system and reported improved system performance after optimization. These studies indicate that normalization and indexing remain relevant and practical approaches in improving database performance, particularly in relational database environments.

Despite the availability of various studies on normalization, indexing, and database tuning, there is still a need for empirical analysis that directly compares database performance before and after optimization using measurable query execution results. Many discussions of normalization focus primarily on data integrity and redundancy reduction, while indexing studies often focus on retrieval speed. However, fewer studies provide a combined analysis of how normalization and indexing work together in improving both data structure and query performance. In addition, the performance effect of these techniques may vary depending on the type of query being executed. SELECT queries may benefit greatly from indexing, while INSERT and UPDATE operations may experience different effects because indexes require additional maintenance during write operations (Gupta et al., 2024; O'Neil et al., 1996; Peterson et al., 2020).

Based on this gap, this study analyzes the implementation of normalization and indexing for database performance optimization using MySQL. The study focuses on three main concerns: how normalization improves data integrity, how indexing affects query performance, and how database performance differs before and after optimization. The experiment compares query execution performance using a database condition before optimization and a database condition after optimization through normalization and indexing. The dataset used in this study consists of 50,000 records, and the performance is evaluated through several query operations, including SELECT, INSERT, and UPDATE. By comparing execution time before and after optimization, this study provides a quantitative basis for understanding the practical impact of normalization and indexing on database performance.

The contribution of this study is both theoretical and practical. Theoretically, this study strengthens the discussion of relational database optimization by integrating normalization and indexing as complementary techniques. Normalization improves the logical structure of data and reduces anomalies, while indexing improves data retrieval

efficiency by supporting faster access paths. Practically, this study provides an implementation-oriented model that can be applied by database designers, system developers, and database administrators in optimizing small to medium-scale information systems. The findings are expected to help practitioners understand that database optimization should not rely on a single technique, but should combine proper schema design, indexing strategy, and query performance evaluation to achieve better database efficiency.

2. Research Methodology

This study employed an experimental research design to evaluate the effect of normalization and indexing on database performance optimization. The experimental approach was selected because the study aimed to compare database performance under two different conditions: before optimization and after optimization. The first condition represented an initial database design that had not been fully normalized and did not implement indexing strategies beyond default key constraints. The second condition represented an optimized database design in which normalization was applied up to the Third Normal Form (3NF), followed by the implementation of indexing on selected attributes. This before–after comparison allowed the study to measure the practical impact of schema improvement and indexing on query execution performance.

The database used in this study was developed using MySQL as the relational database management system. MySQL was selected because it is widely used in web-based applications, academic information systems, business applications, and small to medium-scale information systems. In addition, MySQL provides query optimization features, indexing mechanisms, and execution plan analysis tools that can be used to evaluate database performance. The selection of MySQL was also relevant to the focus of this study, as previous studies have highlighted MySQL performance optimization through indexing, query improvement, and database tuning (Gupta et al., 2024; Ikhwan & Pramadjaya, 2026). The use of a relational database environment was considered appropriate because normalization and indexing are central concepts in relational database design and query processing (Elmasri & Navathe, 2016; Jarke & Koch, 1984).

The experimental dataset consisted of 50,000 records generated using a script-based data generator. The generated data were designed to represent common transactional data in an information system, including identification numbers, names,

categories, dates, status fields, and relational attributes. The use of synthetic data allowed the experiment to control the number of records, data structure, and testing conditions. Although the dataset was not obtained from a real operational system, it was considered sufficient to simulate database performance problems commonly found in small to medium-scale applications. The dataset size of 50,000 records was selected to ensure that query execution time differences could be observed clearly when comparing the non-optimized and optimized database conditions.

The first stage of the research was data requirement analysis. At this stage, the main entities, attributes, and relationships required for the database model were identified. The initial schema was intentionally designed in a less optimal structure to represent common database design problems, such as repeated attributes, redundant records, and potential update anomalies. This condition was used as the baseline for measuring database performance before optimization. The baseline schema allowed the study to observe how unnormalized data structures may affect data consistency, storage efficiency, and query execution.

The second stage was database normalization. The initial database schema was analyzed based on functional dependencies and then normalized up to the Third Normal Form. Normalization was conducted by separating repeated and dependent attributes into related tables, defining primary keys, assigning foreign keys, and reducing unnecessary duplication. The normalization process aimed to improve data integrity, reduce redundancy, and minimize insertion, update, and deletion anomalies. This procedure follows the principles of relational database design, where normalization is used to organize data logically and maintain consistency across related tables (Elmasri & Navathe, 2016; Hardini et al., 2025). In this study, 3NF was selected because it provides a practical balance between data integrity and implementation complexity for transactional information systems.

The third stage was index implementation. After the database schema was normalized, indexes were created on selected columns based on query access patterns. Indexes were applied to primary keys, foreign keys, and frequently searched attributes. In addition, selected columns used in WHERE clauses, JOIN operations, and sorting conditions were considered for indexing. The indexing strategy was designed to improve the efficiency of data retrieval by reducing full-table scans and enabling the database optimizer to use more efficient access paths. This approach is consistent with

the concept of query optimization in relational database systems, where the availability of suitable access paths plays an important role in improving query execution performance (Jarke & Koch, 1984; Selinger et al., 1979). However, the study avoided excessive indexing because unnecessary indexes may increase storage overhead and affect write operations.

The fourth stage was query testing. Several SQL queries were prepared to represent common database operations, including SELECT, INSERT, and UPDATE. SELECT queries were used to test data retrieval performance based on specific conditions, such as searching by ID, searching by name, and retrieving data based on category or status. INSERT queries were used to evaluate the impact of indexing on data insertion performance. UPDATE queries were used to measure the effect of optimization on data modification operations. These query types were selected because they represent basic operations frequently used in information systems. The inclusion of both read and write operations was important because indexing may improve retrieval performance but may also introduce overhead during data insertion and updating (Gupta et al., 2024; O'Neil et al., 1996; Peterson et al., 2020).

The fifth stage was performance measurement. Query performance was measured by recording query execution time before and after optimization. Each query was executed under two database conditions: the non-optimized database and the optimized database. The execution time was recorded in milliseconds. To reduce measurement bias, each query was executed several times, and the average execution time was used as the final measurement value. This repetition was important because execution time may be affected by caching, system load, and temporary resource conditions. The comparison between average execution times was then used to calculate the percentage of performance improvement after optimization.

The percentage of performance improvement was calculated using the following formula:

$$\text{Performance Improvement (\%)} = [(\text{Execution Time Before Optimization} - \text{Execution Time After Optimization}) / \text{Execution Time Before Optimization}] \times 100$$

This formula was applied to each query type to determine the extent to which normalization and indexing improved database performance. A higher percentage indicated a greater reduction in query execution time after optimization. The calculation was particularly useful for comparing the effect of optimization across different query

operations. For example, SELECT operations were expected to show greater improvement because indexing directly supports faster data retrieval, while INSERT operations were expected to show smaller improvement because index structures must be updated when new records are inserted.

In addition to measuring execution time, this study also examined query execution plans using MySQL's EXPLAIN and EXPLAIN ANALYZE features. These tools were used to observe how MySQL processed SQL statements, including the access method, possible indexes, selected indexes, estimated rows, and execution strategy. Execution plan analysis was important because query performance is not determined only by execution time, but also by how the database optimizer selects access paths and processes tables. Cost-based query optimization theory explains that the optimizer evaluates alternative execution plans and selects the plan with the lowest estimated cost based on available statistics, indexes, and query structure (Chaudhuri, 1998; Selinger et al., 1979). Therefore, execution plan inspection was used to support the quantitative execution time results.

The experimental workflow consisted of several systematic steps. First, the data requirements and query scenarios were determined. Second, the initial database schema was created without complete normalization. Third, 50,000 records were inserted into the database. Fourth, query performance was measured in the baseline condition. Fifth, the database schema was redesigned through normalization up to 3NF. Sixth, indexes were applied to selected attributes based on query patterns. Seventh, the same query scenarios were executed again on the optimized database. Finally, the execution time results were compared and analyzed to determine the performance improvement after optimization.

The main variables in this study consisted of independent and dependent variables. The independent variables were normalization and indexing as database optimization techniques. Normalization referred to the restructuring of database tables up to 3NF, while indexing referred to the creation of index structures on selected attributes. The dependent variable was database performance, measured through query execution time. The study also considered query type as an operational comparison factor, including SELECT, INSERT, and UPDATE operations. This variable classification allowed the study to evaluate whether the effect of optimization differed across different types of database operations.

The data analysis technique used in this study was descriptive quantitative analysis. The execution time results from the non-optimized and optimized database conditions were compared using average values and percentage improvement. The analysis focused on identifying differences in performance across query types and explaining why certain operations showed greater improvement than others. The interpretation of results was supported by database optimization theory and previous empirical studies. SELECT queries were expected to benefit most from indexing because indexes reduce the need for full-table scanning. In contrast, INSERT and UPDATE queries were expected to show smaller performance gains or possible overhead because index structures must be maintained during write operations (Gupta et al., 2024; O'Neil et al., 1996).

To ensure the consistency of the experiment, all tests were conducted using the same database management system, dataset size, query scenarios, and testing procedure. The same SQL queries were used before and after optimization to ensure that performance differences were caused by changes in schema design and indexing strategy rather than differences in query structure. The dataset size was also kept constant at 50,000 records in both conditions. These controls were applied to improve the validity of the before–after comparison.

This study was limited to the analysis of normalization and indexing in a MySQL relational database environment. The experiment focused on query execution time as the main performance indicator and did not include other performance dimensions such as memory usage, disk I/O, CPU consumption, concurrency load, or network latency. In addition, the dataset was synthetically generated, so the findings may not fully represent all characteristics of real-world production databases. Nevertheless, the experimental design provides a practical basis for understanding how normalization and indexing affect database performance in small to medium-scale information systems.

3. Results and Discussion

The performance evaluation was conducted by comparing query execution time before and after database optimization. The optimization process consisted of two main stages: normalization of the database schema up to the Third Normal Form (3NF) and implementation of indexes on selected attributes, including primary keys, foreign keys,

and columns frequently used in search conditions. The testing process focused on several common SQL operations, namely SELECT, UPDATE, and INSERT. These operations were selected because they represent the basic database transactions commonly found in small- and medium-scale information systems. The execution time was measured in milliseconds, and the percentage of improvement was calculated by comparing execution time before and after optimization.

The summary of query execution time before and after optimization is presented in Table 1.

Table 1. Query Execution Time Before and After Database Optimization

Query Type	Before Optimization	After Optimization	Performance Change
SELECT by ID	125 ms	15 ms	88.00% faster
SELECT by Name	310 ms	90 ms	70.97% faster
UPDATE	200 ms	80 ms	60.00% faster
INSERT	180 ms	170 ms	5.56% faster

Source: Processed experimental data.

The results show that database optimization produced different performance effects across query types. The highest performance improvement was observed in the SELECT by ID query, where execution time decreased from 125 ms to 15 ms. This represents an 88.00% reduction in query execution time. The SELECT by Name query also showed a substantial improvement, with execution time decreasing from 310 ms to 90 ms, or approximately 70.97% faster than before optimization. These results indicate that indexing provides a strong performance benefit for retrieval operations, particularly when the queried columns are frequently used in search conditions. This finding is consistent with the principle of relational database optimization, where indexes help the database management system locate records more efficiently without scanning the entire table (Elmasri & Navathe, 2016; Gupta et al., 2024; Jarke & Koch, 1984).

The improvement in SELECT operations can be explained by the role of indexes as access structures. Before optimization, the database had to search records through a less efficient access mechanism, which potentially required scanning a large number of rows. After indexes were applied to relevant columns, the database optimizer could use

a more efficient access path to retrieve the required data. This condition supports the concept of access path selection introduced in relational database optimization, where the optimizer evaluates available access paths and selects the most efficient strategy for query execution (Selinger et al., 1979). Chaudhuri (1998) also emphasized that query optimization in relational systems is strongly influenced by schema structure, index availability, database statistics, and the optimizer's ability to estimate execution costs. Therefore, the reduction in execution time found in this study indicates that appropriate indexing can improve the efficiency of query processing in MySQL databases.

The SELECT by ID query showed greater improvement than the SELECT by Name query. This difference may be related to the characteristics of the indexed attributes. ID columns are commonly unique, stable, and often used as primary keys, making them highly efficient for indexed retrieval. In contrast, name-based searches may involve non-unique values, string comparison, or less selective conditions, which may reduce the efficiency of index usage compared with primary key-based searches. This result suggests that the effectiveness of indexing depends not only on whether an index exists, but also on the selectivity of the indexed column, query structure, and the type of comparison used in the SQL statement. Gupta et al. (2024) similarly noted that MySQL performance optimization requires proper index selection because inappropriate or excessive indexing may not produce optimal query performance.

The UPDATE operation also showed a notable improvement after optimization. Execution time decreased from 200 ms to 80 ms, representing a 60.00% reduction. This result indicates that optimization can also support data modification operations when the updated records can be located more efficiently through indexed attributes. In many database transactions, UPDATE statements require the system to first locate specific records before modifying them. When indexes are applied to columns used in the WHERE clause, the database can identify the target records more quickly. As a result, the overall update process may become faster. However, the improvement in UPDATE performance is generally lower than that of SELECT operations because update operations involve not only record retrieval but also data modification and index maintenance. This explains why UPDATE performance improved, but not as substantially as SELECT by ID.

The INSERT operation showed only a small change in execution time, decreasing from 180 ms to 170 ms, or approximately 5.56% faster. This change should be

interpreted cautiously because the improvement is relatively limited compared with SELECT and UPDATE operations. In practical terms, the result indicates that indexing does not substantially accelerate INSERT operations. This is reasonable because every new record inserted into a table may require the database management system to update associated index structures. Index maintenance creates additional overhead during write operations, particularly when a table contains several indexes. O'Neil et al. (1996) explained that storage and index structures affect the way databases handle data insertion and maintenance. Peterson et al. (2020) also showed that transactional data structures and concurrency mechanisms can influence the performance of write operations. Therefore, while indexing is beneficial for data retrieval, its impact on INSERT operations may be limited or even negative if indexes are excessive or poorly designed.

From the perspective of normalization, the optimization process also improved the logical structure of the database. The implementation of normalization up to 3NF helped reduce redundant data and minimized the potential for insertion, update, and deletion anomalies. Although normalization does not always directly reduce query execution time, it improves data integrity and supports better relational design. This is important because database performance should not be understood only in terms of speed, but also in terms of data consistency, maintainability, and storage efficiency. Elmasri and Navathe (2016) emphasized that normalization is a fundamental process in relational database design because it organizes data based on dependencies and reduces undesirable redundancy. Hardini et al. (2025) also found that database normalization can increase storage efficiency by reducing repeated data and improving data organization.

However, normalization may also introduce additional complexity in query execution. When data is separated into several related tables, queries may require join operations to retrieve complete information. Without proper indexing, join operations can increase query execution time and reduce system responsiveness. Therefore, normalization and indexing should be viewed as complementary techniques rather than separate optimization strategies. Normalization improves the structure and integrity of data, while indexing improves access efficiency and query retrieval performance. This relationship is particularly important in relational database systems because a well-normalized schema may still perform poorly if query patterns and index strategies are

not considered during implementation (Jarke & Koch, 1984; Ramakrishnan & Gehrke, 2003).

The combined implementation of normalization and indexing in this study produced the strongest improvement in retrieval-oriented queries. If only the two SELECT operations are considered, the average performance improvement reached approximately 79.49%. If SELECT and UPDATE operations are considered together, the average performance improvement reached approximately 73.00%. However, when INSERT is included in the overall calculation, the average performance improvement decreased to approximately 56.13%. This distinction is important because stating that the overall database performance improved by 70% may be inaccurate if all tested query types are included. A more precise interpretation is that optimization produced approximately 70% or higher improvement for retrieval-oriented and record-locating operations, while the improvement for INSERT operations was relatively limited.

These findings are consistent with previous empirical studies on database optimization. Firdaus et al. (2025) found that relational database optimization through indexing and advanced normalization can improve the performance and structure of academic information systems. Ikhwan and Pramadjaya (2026) also reported that database tuning through indexing and query optimization improved the performance of an academic questionnaire system. Gupta et al. (2024) further explained that indexing is one of the most effective techniques for improving MySQL performance, particularly when queries frequently access large tables. The present study supports these findings by showing that the application of indexing and normalization can substantially reduce query execution time, especially in SELECT operations.

The results also highlight an important practical implication for database designers and system developers. Database optimization should not be performed by adding indexes randomly to all columns. Indexes should be created based on query patterns, column selectivity, relationship structure, and frequency of use in WHERE, JOIN, ORDER BY, and GROUP BY clauses. Excessive indexing may increase storage requirements and reduce write performance because each INSERT, UPDATE, or DELETE operation may require additional index maintenance. Therefore, the optimization process should be based on a balanced strategy that considers both read and write workloads. This finding supports the broader principle of query optimization, which emphasizes that database performance depends on the interaction between schema

design, indexing, query structure, and execution plan selection (Chaudhuri, 1998; Selinger et al., 1979).

In addition, the use of MySQL execution plan analysis is important for evaluating whether indexes are actually used by the optimizer. Execution time alone may show the performance outcome, but execution plan inspection provides deeper information about how the database processes a query. Through tools such as EXPLAIN or EXPLAIN ANALYZE, developers can identify whether a query uses an index, performs a full-table scan, accesses many rows, or applies inefficient join strategies. This type of analysis is useful for validating whether optimization has been implemented effectively. In the context of this study, the reduction in query execution time after indexing suggests that the optimized database design provided more efficient access paths for the tested queries.

Overall, the findings indicate that normalization and indexing contribute to database performance optimization in different but complementary ways. Normalization improves database design quality by reducing redundancy and maintaining data integrity, while indexing improves query execution performance by accelerating data retrieval. The experimental results show that the greatest performance gains occurred in SELECT operations, followed by UPDATE operations, while INSERT operations showed only a limited improvement. These findings suggest that normalization and indexing are highly recommended for small- and medium-scale information systems that experience slow data retrieval and structural data redundancy. Nevertheless, the application of indexing must be carefully planned to avoid unnecessary overhead on write operations.

4. Conclusion

This study demonstrates that normalization and indexing are important techniques for improving the quality and performance of relational database systems. The application of normalization up to the Third Normal Form contributed to a better database structure by reducing data redundancy, minimizing the potential for data anomalies, and improving data consistency. These improvements indicate that normalization plays an essential role not only in organizing data more systematically but also in supporting long-term database maintainability.

The implementation of indexing provided a substantial improvement in query execution performance, particularly for retrieval-oriented operations. The experimental results showed that SELECT by ID achieved an 88.00% reduction in execution time, while SELECT by Name improved by approximately 70.97%. The UPDATE operation also showed a performance improvement of 60.00%, mainly because indexed attributes helped the database locate target records more efficiently. However, the INSERT operation showed only a limited improvement of 5.56%, indicating that indexing does not always provide the same performance benefit for write operations. This finding confirms that indexes are highly effective for read-intensive queries, but they must be implemented carefully because index maintenance can add overhead during data insertion and modification.

Overall, the findings indicate that normalization and indexing should be applied as complementary database optimization strategies. Normalization strengthens the logical structure and integrity of data, while indexing improves data access speed and query execution efficiency. Therefore, the combination of these techniques is recommended for small- and medium-scale information systems that experience problems related to data redundancy, inconsistent records, and slow query response. However, database optimization should not be performed by creating indexes on all columns. Indexes need to be designed based on query patterns, column selectivity, relationship structure, and the frequency of data retrieval operations.

This study provides practical implications for database designers, system developers, and database administrators in improving relational database performance, particularly in MySQL environments. The results suggest that proper schema design and selective indexing can significantly improve system responsiveness without sacrificing data consistency. Nevertheless, this study has several limitations. The experiment used a synthetic dataset consisting of 50,000 records and focused only on SELECT, INSERT, and UPDATE operations. Other performance indicators, such as memory usage, CPU utilization, disk I/O, concurrency load, and network latency, were not examined. Future studies are recommended to use larger and more diverse datasets, test additional query types, evaluate performance under concurrent user access, and compare optimization results across different database management systems such as PostgreSQL, MariaDB, and SQL Server.

References

Budiman, E., Fadli, M., Kurniawan, D., & Susanto, E. R. (2025). Tinjauan literatur dengan pendekatan systematic literature review untuk optimasi query dalam basis data. *Jurnal Ilmiah Teknik dan Ilmu Komputer*. https://www.journal.literasisains.id/index.php/storage/article/view/5182

Chaudhuri, S. (1998). An overview of query optimization in relational systems. *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 34–43. https://doi.org/10.1145/275487.275492

Elmasri, R., & Navathe, S. B. (2016). *Fundamentals of database systems* (7th ed.). Pearson.

Firdaus, U., Rini, A. S., Ghifari, A. A., Saputra, A. A. D., Syahwaludin, M. N., & Aliandy, D. R. (2025). Analisis dan optimasi pada database relasional menggunakan indeks dan normalisasi tingkat lanjut: Studi kasus sistem informasi akademik. *SIGARUDA Journal: Jurnal Ilmiah Bidang Sosial, Ekonomi, Budaya, Teknologi dan Pendidikan, 1*(2), 566–571.

Graefe, G. (1993). Query evaluation techniques for large databases. *ACM Computing Surveys, 25*(2), 73–169. https://doi.org/10.1145/152610.152611

Gupta, M., Mittal, A., Bisht, S. S., & Arora, S. (2024). Optimizing MySQL performance: Essential techniques, resources, and best practices. *2024 International Conference on Progressive Innovations in Intelligent Systems and Data Science (ICPIDS)*, 484–489. https://doi.org/10.1109/ICPIDS65698.2024.00081

Hardini, M., Agarwal, V., Apriani, D., Widjaya, I. A., Setiawaty, E., & Nurasiah, N. (2025). Application of database normalization in increasing data storage efficiency. *International Transactions on Artificial Intelligence, 3*(2), 201–211. https://doi.org/10.33050/italic.v3i2.799

Ikhwan, I., & Pramadjaya, A. (2026). Penerapan database tuning melalui indexing dan optimasi query pada sistem kuesioner akademik: Implementation of database tuning through indexing and query optimization in an academic questionnaire system. *MALCOM: Indonesian Journal of Machine Learning and Computer Science, 6*(2), 716–725. https://doi.org/10.57152/malcom.v6i2.2610

Ioannidis, Y. E. (1996). Query optimization. *ACM Computing Surveys, 28*(1), 121–123. https://doi.org/10.1145/234313.234367

Jarke, M., & Koch, J. (1984). Query optimization in database systems. *ACM Computing Surveys, 16*(2), 111–152. https://doi.org/10.1145/356924.356928

O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996). The log-structured merge-tree (LSM-tree). *Acta Informatica, 33*(4), 351–385. https://doi.org/10.1007/s002360050048

Oracle. (n.d.-a). *MySQL 8.4 reference manual: EXPLAIN statement*. MySQL. Retrieved June 24, 2026, from https://dev.mysql.com/doc/refman/8.4/en/explain.html

Oracle. (n.d.-b). *MySQL 8.4 reference manual: Optimization and indexes*. MySQL. Retrieved June 24, 2026, from https://dev.mysql.com/doc/refman/8.4/en/optimization-indexes.html

Osborn, W. K. (1998). *The use of reduction filters in distributed query optimization* [Master's thesis, Massachusetts Institute of Technology]. MIT DSpace. https://hdl.handle.net/20.500.14776/4296

Özsu, M. T., & Valduriez, P. (2020). *Principles of distributed database systems* (4th ed.). Springer. https://doi.org/10.1007/978-3-030-26253-2

Peterson, C., Wilson, A., Pirkelbauer, P., & Dechev, D. (2020). Optimized transactional data structure approach to concurrency control for in-memory databases. *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 107–115. https://doi.org/10.1109/SBAC-PAD49847.2020.00025

Rahman, M. M., Islam, S., Kamruzzaman, M., & Joy, Z. H. (2024). Advanced query optimization in SQL databases for real-time big data analytics. *Academic Journal on Business Administration, Innovation & Sustainability, 4*(3), 1–14. https://doi.org/10.69593/ajbais.v4i3.77

Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems* (3rd ed.). McGraw-Hill.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access path selection in a relational database management system. *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, 23–34. https://doi.org/10.1145/582095.582099

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database system concepts* (7th ed.). McGraw-Hill Education.