Performance Comparison of RELATIONAL DATABASE MANAGEMENT SYSTEMS

Pang Jen Nee 102771146, Takeru SONODA 102784225

Relational Database Management Systems (RDBMS) serve as the backbone of modern data-driven applications, handling data storage, retrieval, and management. As organizations increasingly rely on data-intensive applications, understanding the performance characteristics of different RDBMS becomes crucial. In this research, we explore the nuances of various RDBMS and compare their performance across critical dimensions to recommend best practices for optimizing performance.

Keywords: ACID Properties, Indexing Strategies, Query Optimization Technologies, Concurrency Control Mechanism, Storage Engine

I. OVERVIEW

Relational databases are foundational in many applications, requiring careful performance evaluation. This review summarises studies comparing different RDBMS focusing on essential concepts like ACID properties, indexing, and query optimisation to understand the literature's strengths, limitations, and gaps.

II. KEY CONCEPTS

A. *ACID Properties*

According to Vithlani(2024), ACID stands for Atomicity, Consistency, Isolation and Durability. ACID properties are an array of crucial properties which assure the integrity of data transactions.

Atomicity ensures that each command in a transaction is handled as a single unit and that they all either succeed or fail together. This is necessary because it allows the users to be certain of the database's condition in the event of an unintentional incident, such as a crash or power outage. If any portion of the transaction failed, it would have either been successfully completed or rolled back (What are ACID Properties in Database Management Systems? 2024)

Consistency ensures that changes which are done during a transaction are to be compatible with database constraints. This comprises of all guidelines, limitations, and triggers. If the data gets into an illegal state, the entire transaction fails.

With Isolation, all transactions are guaranteed to occur in an isolated environment. As transactions don't conflict with one another, this makes it possible to conduct multiple transactions at once.

Durability ensures that changes are preserved in the database after the transaction is over. This guarantees that, even in the event of system failures such as crashes or power outages, data within the system will remain intact.

These properties ensure data reliability but implementing them may reduce performance under high load due to validation checks and locking.

B. *Indexing Strategies*

According to Alooba, indexing strategies cover various techniques for organizing and structuring management databases (Alooba c. 2024). This includes exploring different types of indexes, such as B-Tree indexing, hash indexing, bitmap indexing, clustered indexing, and covering indexing. Each technique offers unique ways to optimize data retrieval and improve database performance. B-Tree is ideal for both range and equality queries due to hierarchical data arrangement. Hash is effective for exact-match queries by mapping data to unique identifiers. Bitmap is efficient for low-cardinality data by creating bitmaps of possible values. Clustered physically organises data for faster retrieval in range queries. Finally, Covering reduces table access by storing all relevant columns directly in the index.

C. *Query Optimization Techniques*

A key component of database management systems (DBMS) is query optimization, which looks at several query execution methodologies to find the most effective way to execute a particular query. Reducing the amount of system resources required to respond to the query and speeding up the outcome of responses are the objectives (Dremio c. 2024).

Fundamentally, query optimization involves comparing various query plans and selecting the one with the lowest projected cost. Key features of query optimization include parsing, which converts SQL queries into a query tree; transformation, which uses optimization, normalization, and simplification to improve the query tree; cost estimation, which calculates how much each possible execution plan would cost; and plan selection, where the most cost-effective plan is chosen and implemented by the DBMS.

D. *Concurrency Control Mechanism*

Concurrency control mechanism in a database makes sure that concurrent transactions don't conflict with one another. Additionally, it avoids frequent problems like non-repeatable reads, missing updates, and dirty reads. The three main functions are preventing deadlocks, guaranteeing transaction serializability, and preserving data integrity (Dremio c. 2024).

There are two main types of concurrency control protocols: Optimistic Concurrency Control (OCC) and Pessimistic Concurrency Control (PCC). OCC assumes minimal conflict and performs checks at the end, allowing the same database item to be accessed or updated by many transactions without obtaining locks, making it suitable for high-read systems (Adetunji, 2024). PCC, on the other hand, prevents conflicts by locking resources until the transaction is complete, ensuring correct serialization. Both methods handle high transaction volumes efficiently and scale well.

E. *Storage Engine*

According to the Sisense Team (2024), different storage engines serve specific needs in MySQL. InnoDB offers transaction support and ACID compliance, making it suitable for high-reliability applications. On the other hand, MyISAM is designed for read-heavy environments that do not require transaction support. Choosing the right engine is crucial for achieving optimal performance and scalability.

III. Methodology

A. *Database Selection*

This study compares the performance of three commonly used relational database management systems (RDBMS) which are MySQL, PostgreSQL, and MariaDB.

- MySQL is chosen due to popularity in web applications and is known for its reliability and high transactional speed when dealing with diverse data workloads.
- PostgreSQL is selected due to its powerful features and efficiency in managing complicated queries and large scale datasets, making it the best choice for at company level applications.
- MariaDB, almost same as MySQL, is included to object if its specific optimizations can boost any performance compared to processor.

These databases were selected to provide a broad perspective on the performance and capabilities of both open-source and commercial RDBMS under various conditions.

B. *Benchmarking Metrics and Workload Scenarios*

To assess the performance of each database system, the following metrics will be analysed

- Response Time measures the time taken to complete a query, reflecting the speed of data retrieval.
- Throughput evaluates the number of transactions processed per unit of time, indicating the efficiency of the database under load.
- Scalability assesses the database's ability to maintain performance levels as the size of the dataset and number of concurrent users increase.

Workload scenarios to be tested include

- Read-intensive Workload simulates a scenario dominated by read operations, typical in data analysis and reporting applications.
- Write-intensive Workload focuses on write operations relevant to transactional systems such as online transaction processing (OLTP) systems.

C. *Experimental Setup*

Hardware Configurations:

- Processor: AMD Ryzen 7 6800H with Radeon Graphics, 3.20 GHz
- Installed RAM: 16.0 GB

Software configurations:

Operating System: Windows with Windows Subsystem for Linux (WSL)

Database Configurations

- MySQL (ver 8.0.36): Installed on WSL, using an online dataset 'employees.sql' from https://github.com/datacharmer/test_db.
- MariaDB (ver 11.3.2): Installed in a Docker container on WSL, using the employees_partitioned.sql dataset and set to 3307.
- PostgreSQL: Optimised and installed on WSL, trying setting up the database from SQL server to PostgreSQL using pgloader, but it failed, using a custom Python script to setup and migrate the database.

Tool Used

- Sysbench: For overall database performance benchmarking.
- Pgloader; Data loading tool for PostgreSQL which can migrate the whole database when needed.
- Python Scripts: Custom scripts to execute specific queries and measure performance.

IV. Performance Comparison

A. *Query Excution Time*

To evaluate the query execution time, the following queries were executed on each database.

1st Query SELECT * FROM employees WHERE emp_no = 10001

2nd Query SELECT COUNT(*) FROM salaries WHERE salary > 50000

Query	MySQL Execution Times(ms)	MariaDB Execution Times(ms)	PostgreSQL Execution Times(ms)

SELECT * FROM employees WHERE emp_no = 10001	0.507	0.467	1.053
SELECT COUNT(*) FROM salaries WHERE salary > 50000	289.449	633.533	94.435

Table 1:Comparison of Query Execution Time

B. Indexing Impact

To evaluate query performance with and without indexes, indexes were created to evaluate their impact:

“CREATE INDEX idx_emp_no ON employees(emp_no)”,

“CREATE INDEX idx_salary ON salaries(salary)”

The queries evaluated were the same as those used in the Query Execution Time section:

Before Index

Database	Query1(ms)	Query2(ms)
MySQL	0.534	261.237
MariaDB	0.484	606.877
PosgreSQL	1.107	84.741

Table 2: Before Indexing Database

After Index

Database	Query1(ms)	Query2(ms)
MySQL	1.052	333.671
MariaDB	1.057	620.910
PosgreSQL	1.456	70.700

Table 3: After Indexing Database

C. Query Optimization Techniques

The query execution plans were analyzed by using the same queries from the query execution time experiment.

1) *MySQL Execution Plans*

SELECT * FROM employees WHERE emp_no = 10001
Plan: [(1, 'SIMPLE', 'employees', None, 'const', 'PRIMARY,idx_emp_no', 'PRIMARY', '4', 'const', 1, 100.0, None)]

SELECT COUNT(*) FROM salaries WHERE salary > 50000
Plan: [(1, 'SIMPLE', 'salaries', None, 'range', 'idx_salary', 'idx_salary', '4', None, 1419213, 100.0, 'Using where; Using index')]

2) *MariaDB Execution Plans*

SELECT * FROM employees WHERE emp_no = 10001
Plan: [(1, 'SIMPLE', 'employees', 'const', 'PRIMARY,idx_emp_no', 'PRIMARY', '4', 'const', '1', '')]

SELECT COUNT(*) FROM salaries WHERE salary > 50000
Plan: [(1, 'SIMPLE', 'salaries', 'range', 'idx_salary', 'idx_salary', '4', None, '2834326', 'Using where; Using index')]

3) *PostgreSQL Execution Plans*

SELECT * FROM employees WHERE emp_no = 10001
Plan: [('Index Scan using employees_pkey on employees (cost=0.42..8.44 rows=1 width=29)',), (' Index Cond: (emp_no = 10001)',)]

SELECT COUNT(*) FROM salaries WHERE salary > 50000
Plan: [('Finalize Aggregate (cost=33449.48..33449.49 rows=1 width=8)',), (' -> Gather (cost=33449.26..33449.47 rows=2 width=8)',), (' Workers Planned: 2',), (' -> Partial Aggregate (cost=32449.26..32449.27 rows=1 width=8)',), (' -> Parallel Seq Scan on salaries (cost=0.00..30186.74 rows=905008 width=0)',), (' Filter: (salary > 50000)',)]

D. Concurrency Control and Locking

The graph below illustrates the response times for MySQL, MariaDB and PostgreSQL as the number of concurrent transactions increases from 1 to 10.

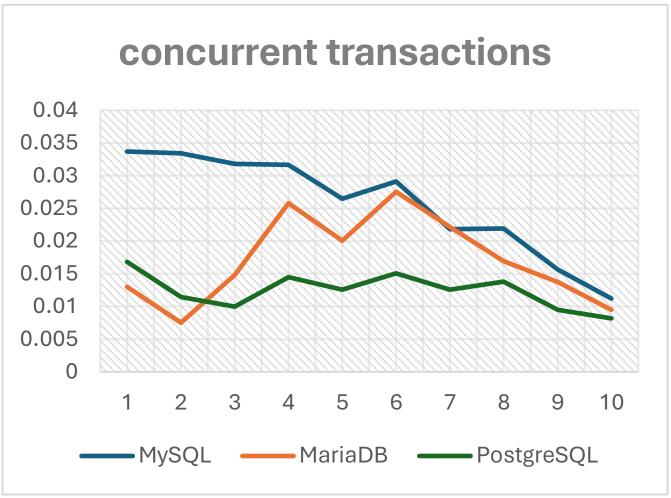

Table 4: Response Times vs Concurrent Transactions

E. Storage Engine Comparison

Assessment of the impact of different storage engines (if applicable):

MySQL using InnoDB

- Insert Time: 1701.41 ms
- Features: Supports transactions and ACID compliance for high-reliability applications

MySQL MyISAM

- Insert Time: 15400.22 ms
- Features: Optimised for read-heavy environments, lacks transaction support.

MariaDB InnoDB

- Insert Time: 1890.01 ms
- Features: Similar to MySQL InnoDB, with transaction support and ACID compliance.

MariaDB MyISAM

- Insert Time: 1876.60 ms
- Features: Similar to MySQL MyISAM, optimised for reads, no transaction support.

V. Result and Discussion

A. Query Execution Time Results

As observed from the table of results, the execution time for the first query is the fastest for MariaDB. This suggests that MariaDB handles simple primary key lookups efficiently.

However, for the second query, PostgreSQL takes a clear lead in speed. This indicates that PostgreSQL is very efficient in handling aggregation queries.

B. Index Impact Results

The two queries used in testing the indexing impact represents two scenarios. Query1 performs an equality search which represents a common use case where the user searches for a specific record based on a unique identifier. Query2 involves an aggregation function (COUNT(*)) and a range condition (salary > 50000).

In the case of Query1, indexing didn't lead to improvements in performance in all the databases. As it's a simple query which is efficient even without indexing. Indexing primarily helps with range queries, joins, and more complex conditions. Since Query1 is straightforward, the impact of indexing is less pronounced.

However, in Query2, PostgreSQL is seen to be the only database with an improvement in efficiency with a huge difference of speed compared to the MySQL and MariaDB. It might be since PostgreSQL has a sophisticated parallel execution framework.

C. Query Optimization Techniques Results

- MySQL Analysis
  - SELECT * FROM employees WHERE emp_no = 10001
    - A simple primary key search, using the 'const' access method. It uses the PRIMARY key and the 'idx_emp_no' index.
  - SELECT COUNT(*) FROM salaries WHERE salary > 50000
    - The use of the 'range' access type with the 'idx_salary' index demonstrates efficient handling of range queries, though there is room for optimization.
- MariaDB Analysis
  - SELECT * FROM employees WHERE emp_no = 10001
    - Like MySQL, MariaDB efficiently handles primary key lookups with the 'const' access type, effectively using the primary key and 'idx_emp_no' index for efficient retrieval.
  - SELECT COUNT(*) FROM salaries WHERE salary > 50000
    - It uses the 'range' access method and leverages the 'idx_salary' index to efficiently process range queries.
- PostgreSQL Analysis
  - SELECT * FROM employees WHERE emp_no = 10001
    - Uses an 'Index Scan' for primary key retrieval efficiency.
  - SELECT COUNT(*) FROM salaries WHERE salary > 50000
    - The parallel sequential scan and aggregation highlight PostgreSQL's strength in handling complex queries.

D. Concurrency Control and Locking Results

MySQL starts with relatively high response times (around 0.035 seconds) for a single concurrent transaction. As the number of concurrent transactions increases, MySQL's response times slightly decrease but remain relatively stable overall. MySQL handles concurrency relatively well but shows longer response times compared to other databases.

MariaDB begins with the lowest initial response times (around 0.015 seconds) for a single transaction. Interestingly, as the number of concurrent transactions increases, MariaDB's response times initially increase but then improve. MariaDB's performance even surpasses MySQL's at certain points.

PostgreSQL consistently exhibits the lowest response times across all levels of concurrency, with the initial response time at around 0.01 seconds. Unlike MySQL and MariaDB, PostgreSQL's response times do not show significant fluctuations. PostgreSQL maintains its efficiency even as the workload increases, demonstrating very stable performance.

E. *Storage Engine Comparison Results*

1) MySQL InnoDB vs. MyISAM

The results indicate that MySQL with InnoDB has a significantly faster insert time compared to MyISAM. InnoDB's faster insert performance suggests that it is better suited for applications with higher write loads. MyISAM's much slower insert time indicates that it may not be ideal for write-heavy applications, though it might still be useful in scenarios where read performance is more critical.

2) MySQL InnoDB vs. MariaDB InnoDB

Similarly, MariaDB with InnoDB shows a faster insert time than MyISAM. This reinforces the idea that InnoDB is more suitable for applications requiring efficient write operations. The slower insert time for MariaDB with MyISAM suggests it is less effective for write-heavy tasks.

3) MySQL InnoDB vs. MariaDB MyISAM

Comparing the InnoDB storage engines between MySQL and MariaDB, MySQL shows a slightly faster insert time. This small difference may be due to implementation details or optimizations specific to each database system.

4) MySQL InnoDB vs. MariaDB MyISAM

For the MyISAM storage engine, MariaDB has a slightly faster insert time compared to MySQL. Although both engines are slower than their InnoDB counterparts, the performance difference between MySQL and MariaDB's MyISAM is minimal.

VI. Recommendations and Best Practices

A. *Database Selection*

For read-heavy applications, further testing is required to confirm the suitability of MariaDB with the MyISAM storage engine. While MyISAM is generally considered to favor read operations due to its design, our experiments focused solely on insert times. Therefore, its read performance should be specifically measured to make a definitive recommendation. Based on the insert time results, InnoDB demonstrated significantly faster insert times, making it more suitable for write-intensive applications.

For write-intensive applications, MySQL or MariaDB with the InnoDB storage engine is preferable. The experiments showed that InnoDB had significantly faster insert times compared to MyISAM, indicating its suitability for environments with high write loads. InnoDB also supports transactions and is ACID-compliant, ensuring high reliability and data integrity.

PostgreSQL is recommended for handling complex queries and large datasets. The experiments indicated that PostgreSQL performs very well with aggregation queries, likely due to its advanced parallel processing capabilities. This makes PostgreSQL an ideal choice for data analytics platforms and enterprise applications that require robust data manipulation and querying.

B. *Guidelines for Optimazing Performance*

1) Indexing Strategies

Adding indexes is crucial for enhancing database performance, particularly for complex queries and aggregations. In the experiments, PostgreSQL showed significant performance improvements with indexing for complex queries. While unnecessary indexes should be avoided for simple queries to prevent overhead, they can significantly improve the performance of range queries and aggregations in PostgreSQL. It is important to periodically update and maintain indexes to ensure they remain optimized for current data patterns. For other databases, specific testing is recommended to confirm the impact of indexing.

2) Query Tuning

Regular analysis and tuning of query execution plans are essential for identifying and addressing bottlenecks and inefficiencies. The experiments highlighted the effectiveness of PostgreSQL's parallel query execution and cost-based optimization for enhancing query performance. Rewriting queries to optimize subqueries and join operations can lead to significant performance improvements. However, these techniques should be tested and confirmed for other databases as well, as their specific optimization features may vary.

3) Concurrency Control and Locking

Effective concurrency control mechanisms are vital for applications with high transaction volumes. PostgreSQL demonstrated stable and low response times under concurrent transactions, indicating its robust concurrency control mechanisms. Using row-level locking can minimize contention in highly concurrent environments. Optimizing transaction design to keep transactions short and reducing lock duration can also improve performance. Implementing appropriate isolation levels helps balance performance and data consistency requirements. These principles should be applied and tested in the context of other databases to ensure their effectiveness.

4) Storage Engine

The choice of storage engine significantly impacts performance. InnoDB demonstrated faster insert times compared to MyISAM in the experiments, indicating it may be more suitable for applications with higher write loads. MyISAM, on the other hand, showed slower insert times, suggesting it may not be ideal for write-heavy applications. When choosing the appropriate storage engine, it is important to consider the specific needs and workload characteristics of the application. InnoDB should be chosen for applications that require efficient write operations and data integrity. By carefully considering the application's specific workload characteristics, you can choose the storage engine that best fits the application's needs, maximizing performance and efficiency.

VII. Conclusion

This research underscores the critical importance of understanding database performance characteristics for application development. Each database system exhibits unique strengths and weaknesses, making them suitable for different application scenarios. Based on the insert time results, MariaDB with the MyISAM storage engine may be suitable for read-heavy applications, although further testing is required to confirm this. MySQL or MariaDB with the InnoDB storage engine is better suited for write-intensive applications due to significantly faster insert times. PostgreSQL excels in handling complex queries and large datasets, as evidenced by its performance in aggregation queries.

By selecting the appropriate database and employing effective optimization techniques such as indexing strategies, query tuning, and proper concurrency control, developers can significantly enhance application performance. The study highlights the importance of tailoring the database system to the specific needs of the application, ensuring that the chosen solution aligns with operational demands.

Future research could explore the performance of NoSQL databases and hybrid models, providing a broader perspective on database performance optimization. Additionally, investigating the impact of different hardware configurations and real-world workloads could further refine our understanding of database performance. Integrating machine learning techniques for predictive performance tuning could also be a promising area for future studies.

Understanding and applying these recommendations and best practices will enable developers to optimize database performance effectively, ensuring robust and reliable software development outcomes.

VIII. References

Adetunji, H 2024, *How Databases Guarantee Isolation – Pessimistic vs Optimistic Concurrency Control Explained*, freeCodeCamp, viewed 21 May 2024, <https://www.freecodecamp.org/news/how-databases-guarantee-isolation/#optimistic-concurrency-control>.

Alooba c. 2024, *Indexing Strategies,* Alooba, viewed 20 May 2024, <https://www.alooba.com/skills/concepts/database-and-storage-systems/database-management/indexing-strategies/#:~:text=In%20simple%20terms%2C%20indexing%20strategies,quickly%20locate%20specific%20data%20entries.>.

Dremio c. 2024, *Concurrency Control,* Dremio, viewed 21 May 2024, <https://www.dremio.com/wiki/concurrency-control/#:~:text=Concurrency%20Control%20ensures%20that%20simultaneous,transactions%2C%20and%20maintaining%20data%20integrity.>.

Dremio c. 2024, *Query Optimization,* Dremio, viewed 20 May 2024, <https://www.dremio.com/wiki/query-optimization/>.

Sisense Team 2024, ‘The Beginner’s Guide to MySQL Storage Engines’, *Sisense*, viewed 21 May 2024, <https://www.sisense.com/blog/beginners-guide-to-mysql-storage-engines/>.

Vithlani, H 2024, ‘ACID Properties in DBMS’, 30 April, viewed 19 May 2024, <https://www.scaler.com/topics/dbms/acid-properties-in-dbms>.

What are ACID Properties in Database Management Systems?, 2024, MongoDB, viewed 19 May 2024, < https://www.mongodb.com/resources/basics/databases/acid-transactions#:~:text=of%20unexpected%20errors.-,Atomicity,the%20state%20of%20the%20database.>.