future internet

MDPI

Review

A Performance Benchmark for the PostgreSQL and MySQL Databases

Sanket Vilas Salunke and Abdelkader Ouda *

Department of Electrical and Computer Engineering, Western University, London, ON N6A 5B9, Canada; ssalunke@uwo.ca

* Correspondence: aouda@uwo.ca

Abstract: This study highlights the necessity for efficient database management in continuous authentication systems, which rely on large-scale behavioral biometric data such as keystroke patterns. A benchmarking framework was developed to evaluate the PostgreSQL and MySQL databases, minimizing repetitive coding through configurable functions and variables. The methodology involved experiments assessing select and insert queries under primary and complex conditions, simulating real-world scenarios. Our quantified results show PostgreSQL's superior performance in select operations. In primary tests, PostgreSQL's execution time for 1 million records ranged from 0.6 ms to 0.8 ms, while MySQL's ranged from 9 ms to 12 ms, indicating that PostgreSQL is about 13 times faster. For select queries with a where clause, PostgreSQL required 0.09 ms to 0.13 ms compared to MySQL's 0.9 ms to 1 ms, making it roughly 9 times more efficient. Insert operations were similar, with PostgreSQL at 0.0007 ms to 0.0014 ms and MySQL at 0.0010 ms to 0.0030 ms. In complex experiments with simultaneous operations, PostgreSQL maintained stable performance (0.7 ms to 0.9 ms for select queries during inserts), while MySQL's performance degraded significantly (7 ms to 13 ms). These findings underscore PostgreSQL's suitability for environments requiring low data latency and robust concurrent processing capabilities, making it ideal for continuous authentication systems.

Keywords: benchmark; postgreSQL; mySQL; database; biometrics

Citation: Salunke, S.V.; Ouda, A. A Performance Benchmark for the PostgreSQL and MySQL Databases. *Future Internet* **2024**, *16*, 382. https://doi.org/10.3390/fi16100382

Academic Editors: Michael Sheng, Yanhao Wang and Chengcheng Yang

Received: 30 September 2024
Revised: 16 October 2024
Accepted: 18 October 2024
Published: 19 October 2024

1. Introduction

In the digital world, latency is the new outage. Simply put, latency means delay. In technological terms, it is the time required to perform any action or operation. For example, when a user searches on Google, the search engine takes time to display all related results. The time difference between entering the query and obtaining the result is called latency. It is essential to study the latency of a system, as it has a major impact on performance. In the case of continuous user authentication, reducing latency is critical as the users are authenticated on an ongoing basis; any latency can create an opportunity for hackers. As a rule of thumb, lower latency equates to higher speed and performance. Therefore, it is crucial to identify and reduce latency. One significant factor causing lag is database performance, also called data latency, which is the time taken to store and retrieve data from the database. In continuous authentication [1,2], users' raw data are collected and saved in the database (as shown in Figure 1). Here the insert operation is performed, and later, these data are fetched for feature extraction.

Data latency is key, especially when large amounts of data are added or fetched, such as in continuous user authentication, where around 50 new raw behavioral data records are generated and stored in a database every second for each user. Considering there are a few hundred users, this would generate millions of records every day, consequently impacting database performance. Hence, it is essential to select the appropriate database using the database benchmarking technique.

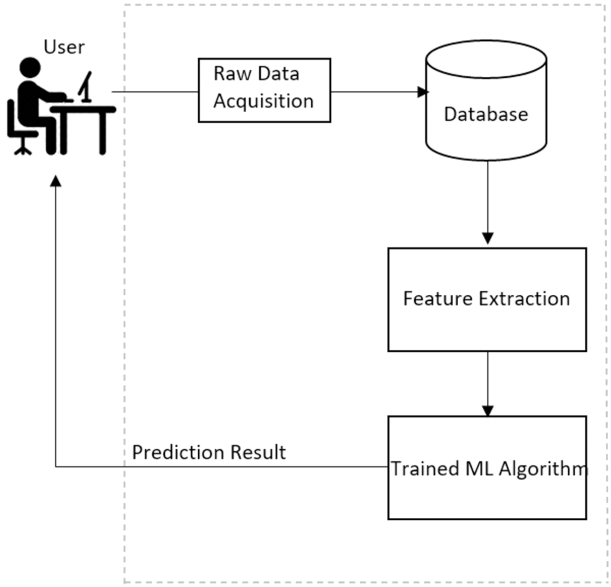

Figure 1. Continuous authentication architecture.

Database benchmarking is a well-defined process of evaluating, measuring, and comparing the performance of different database systems to assess multiple aspects of a database, including its scalability, speed, efficiency, and ability to handle transactions or queries under various conditions [3–5]. Generally, database operations such as insert, delete, read, and update have different workloads that affect database latency. Therefore, it is significant to conduct benchmarking before installing a database system into an application.

Database benchmarking is a critical tool to ensure that the chosen database meets expectations regarding handling current and future loads efficiently [6,7]. The benchmarking test provides a basis for comparing various databases and helps provide insights for database professionals on the suitability of various database management systems. As data grow, so does the workload handled by their database system; hence, conducting benchmarking tests initially helps prevent future problems in long-term capacity planning. Benchmarking provides measurable metrics like query execution time and transaction throughput, ensuring that the chosen database meets specific application needs and handles real-world demands.

The relational database management system (RDBMS) is a broadly accepted system that can create meaningful information by joining tables with SQL (Structural Query language), which is most commercially used to access databases. The RDBMS platform provides a dependable method of storing and retrieving large amounts of data and has good performance, scalability, and reliability. It is foundational for applications across industries like finance, healthcare, manufacturing, etc., due to its lifespan, forming a large community around it that helps improve overall system needs.

PostgreSQL [8] and MySQL [9] are two prominent open-source RDBM systems known for their performance robustness and adoption across industries: PostgreSQL for its advanced features supporting concurrent transactions, and MySQL for its high-speed transactions—making benchmarking comparisons between them meaningful, as both serve diverse application needs, allowing objective performance comparison under real-world workloads. Studying execution times under varying conditions is crucial because databases directly impact application performance, to ensure that databases are continuously tailored to users' authentication needs.

There are many research contributions on database benchmarking; however, research gaps exist where they remain generic and not specifically tailored towards continuous user

authentication scenarios. This study focuses on creating frameworks reflecting real-world conditions of continuous user authentication before presenting a proposed solution and research questions and providing recent literature summaries highlighting the importance of addressing these gaps:

- Continuous Authentication Challenges: Continuous authentication systems face unique challenges such as fast data updates and efficient query processing, which are critical for minimizing latency [2,10]. These challenges emphasize the need for specialized benchmarking frameworks that can evaluate database performance under such demanding conditions.
- Importance of Database Performance: Studies have shown that efficient database management is crucial in scenarios involving large volumes of data, such as continuous user authentication [11]. Our proposal's focus on data latency aligns with findings that highlight the impact of database performance on overall system efficiency.
- Innovative Approaches: New developments in continuous authentication technologies, like using behavioral biometrics and machine learning models, make it even more important to have strong backend systems that can handle a lot of data coming in quickly [12,13]. This further supports the necessity for a tailored benchmarking approach such as that proposed in this study.

In this study, we propose a novel benchmarking database considering the following key research questions:

- What are the factors contributing to latency in continuous user authentication systems, and how can they be minimized?
  This study will highlight the importance of understanding latency, particularly data latency, in systems where continuous user authentication is critical. This involves identifying specific causes of latency in database operations and exploring methods to mitigate them.
- Which database engine is most suitable for continuous user authentication applications, considering the need for low latency?
  This study proposes developing a universal benchmarking framework to evaluate databases under practical conditions similar to those found in continuous user authentication scenarios. This involves comparing PostgreSQL and MySQL to determine which database engine better handles structured and tabular data efficiently.
- How can a benchmarking framework be designed to evaluate database performance in production-like scenarios for continuous authentication?
  This research intends to create a Python-based framework that benchmarks databases based on various operations typical in continuous user authentication environments. This framework aims to identify the best database fit for such applications and can be adapted to other databases with minimal code changes.

The remaining sections of this paper are arranged as follows: A summary of the PostgreSQL and MySQL technologies will be given in Section 2, and a thorough explanation of the suggested benchmarking framework and the experimental setup will be given in Section 3. The framework experiments and their findings are covered in detail in Section 4, and our concluding thoughts are provided in Section 5.

2. Overview of PostgreSQL and MySQL

PostgreSQL is derived from the POSTGRES package developed at the University of California at Berkeley [14]. It was led by Professor Stockbroker and sponsored by the Defense Advanced Research Projects Agency (DARPA). POSTGRES has undergone many releases since 1987. In 1994 [15], Andrew Yu and Jolly Chen added the SQL language to POSTGRES and released it as an open-source descendant of the original POSTGRES Berkeley code. By 1996 [15], PostgreSQL was chosen to reflect the relationship between SQL and the original POSTGRES. After around 35 years of its release, it has become one of the most popular database choices for many large organizations like Apple, Reddit,

and Instagram to store data. The latest version of PostgreSQL focuses on performance, parallelism, and cloud-native deployment support. It has a wide range of data types, including standard SQL types, arrays, JSON/JSONB for semi-structured data, and geospatial data through the PostGIS extension. PostgreSQL excels in its implementation of Multi-Version Concurrency Control (MVCC) and advanced indexing techniques [16,17]. MVCC allows concurrent transactions to access the same data without blocking each other, a feature that is crucial for maintaining high performance in environments with many simultaneous users. Transaction isolation ensures that transactions do not interfere with each other, providing consistent and isolated views of the data. PostgreSQL also uses vacuuming to manage storage overhead caused by multiple row versions. PostgreSQL offers various indexing options to enhance its ability to perform complex queries efficiently. These include B-tree [18], GiST (Generalized Search Tree) [19], GIN (Generalized Inverted Index) [20], and BRIN [21], each optimized for different types of queries. Partial and expression indexes allow for the creation of partial indexes that index a subset of rows based on a specified condition, while concurrent index creation allows read and write operations to continue during index creation, particularly beneficial in high-concurrency environments.

On the other hand, MySQL has established itself as one of the most popular and widely used databases in the industry and is known for its speed and reliability. MySQL was developed by Michael Widenius, David Axmark, and Allan Larsson in 1995 at the Swedish company MYSQL AB [22]. In 2000, MySQL became open-source and gained popularity. By 2001, it had over 2 million active users and was later acquired by Oracle Corporation. MySQL is particularly popular for web applications and is the backbone of many online platforms, including Facebook, Twitter, and YouTube. It supports standard SQL and offers features like replication, partitioning, and full-text search. MySQL is known for its performance in read-heavy environments and is widely used in LAMP (Linux, Apache, MySQL, PHP) stacks, making it a top choice for developers and businesses globally.

In the past, multiple studies have been conducted on database benchmarking. In one such study [4] the performance of key-value databases (Redis and Memcached) was compared with that of relational databases (MySQL and PostgreSQL) using the Yahoo! Cloud Serving Benchmark (YCSB). This benchmarking focuses on running time, throughput, and latency under varying workloads (A, C, F) with different operations and thread counts. Key-value databases like Redis excelled in performance, demonstrating lower run times, higher throughput, and reduced latency, especially for workloads requiring high concurrency. Redis consistently outperformed the other databases, particularly in multi-thread scenarios, while Memcached faced issues with larger update operations. In contrast, relational databases generally showed poorer performance compared to NoSQL alternatives, despite improving with more threads. MySQL had the weakest results overall. The study concluded that NoSQL databases are better suited to handling large datasets with higher efficiency, while relational databases like MySQL and PostgreSQL still retain their relevance in structured-data scenarios.

In another study conducted in 2021 [23], the performance of Redis, MongoDB, and Cassandra is compared using the Yahoo Cloud Serving Benchmark (YCSB) across various workloads, including read, write, scan, and update operations. Redis outperforms the other databases in read-heavy tasks due to its use of volatile memory, making it the fastest for read operations. MongoDB excels in write-heavy and scan-intensive workloads, showing superior efficiency in data loading and short-range scans. In contrast, Cassandra lags in both read and write tasks, with the slowest overall performance. Their study highlights Redis as best suited for read-heavy workloads, while MongoDB is preferable for write-heavy scenarios, with Cassandra being the least optimized of the three.

Filip P and Cegan L [24] conducted a study comparing the performance of MySQL and MongoDB in web applications. Their study highlights the limitations of traditional SQL databases, which struggle with scaling, and contrasts this with the flexibility of NoSQL systems like MongoDB. Their experiments focus on common database operations (insert, update, delete) in both transactional and non-transactional modes. MongoDB consistently

outperforms MySQL, especially in transactional queries. MySQL's indexing improves search performance but significantly slows down data modification tasks. Their findings suggest that MongoDB's scalability and performance make it a strong choice for applications where speed is crucial.

The benchmarking study [25] conducted in 2016 compared the performance of two popular open-source databases, MySQL and MariaDB [26], focusing on their efficiency in handling workloads in a virtualized environment using Xen 4.4 [27]. The researchers employed OLTP-Simple and OLTP-Seats [28], which simulate real-world online transaction processing (OLTP) workloads, such as flight booking systems. They tested the databases with varying levels of threads and workers to observe resource utilization in terms of CPU and memory consumption. Their key findings showed that MySQL outperformed MariaDB in high-thread environments, particularly with 1,000 threads in OLTP-Simple and four workers in OLTP-Seats. Both databases showed similar CPU and memory usage, but MySQL consistently demonstrated higher transaction throughput. Future research aims to explore MariaDB's performance in cluster environments.

3. Materials and Methods

A benchmarking framework was built to simplify the development environment and avoid coding repetitive functions and libraries. This framework can be used to benchmark different databases with minimal modification to a few functions and configuration variables. Figure 2 gives a brief overview of the framework.

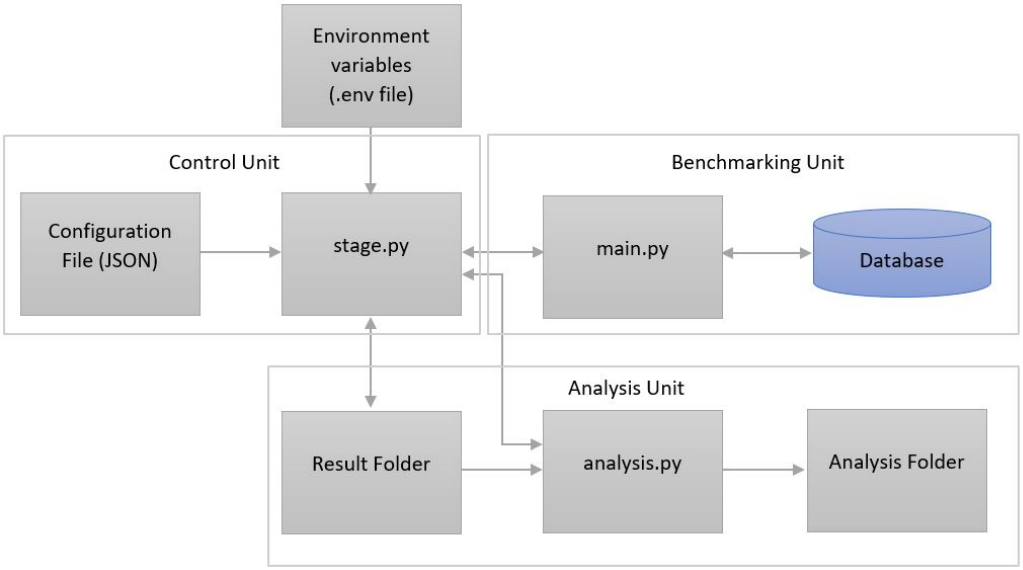

Figure 2. Benchmarking framework block diagram.

A detailed description of each component is given below.

3.1. Control Unit

This unit has the configuration, environment, and stage file necessary to begin the benchmarking experiments. Details of each file are described below.

- Configuration file: This is a JSON file where different configuration variables can be set based on the experiment. It has a total of 23 properties, which are of the String, Array, and Boolean types. The file's details are shown in Figure 3 below.

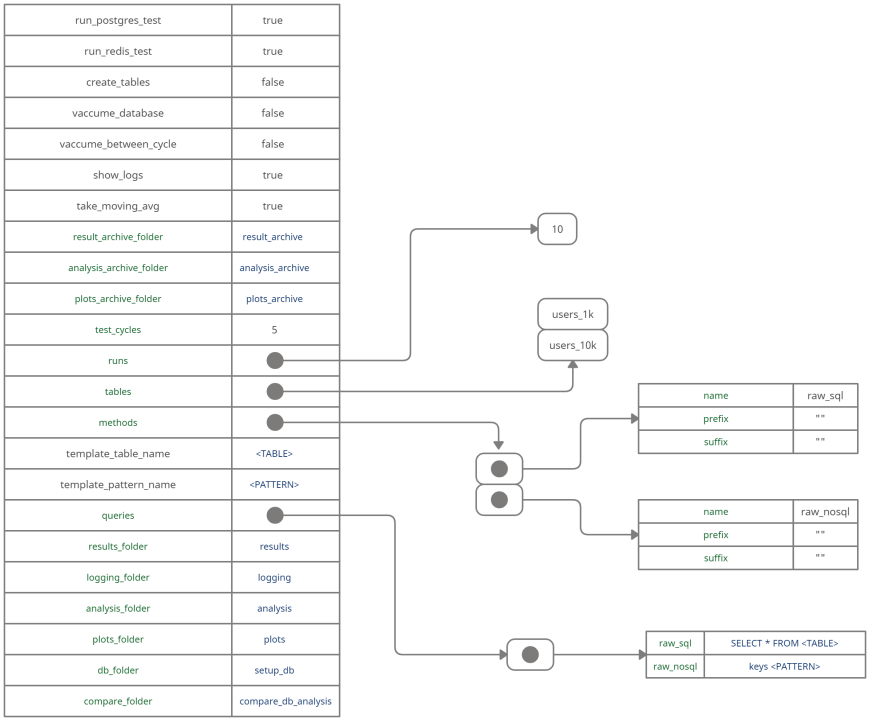

Figure 3. Configuration file variables.

The configuration file has more than 40 keys, but some important keys are explained below.

- runs: This is an array that determines how many times the query should run. For instance, if the value of a run is an array of values 10 and 20, then the query would run 10 times and then 20 times.
- test_cycles: This key has an integer value that defines how many times the test should be executed for the runs. For example, if the value of runs is 10 and test_cycles is 5, then there will be five cycles with 10 runs in each cycle.
- methods: This key has a nested array of an object that defines the name of the method, suffix, and prefix.

The suffix and prefix can be used to build the query for the method.

- queries: This is a nested JSON object that has queries used for benchmarking the database. Here, the method name is used to identify the query.
- Environmental variables (.env file): This file has environment variables required to connect to the database, such as the database server IP address, port number, username, and password. Each database has an individual environment file to store these data, which is then used to make the database connection.
- Stage file: This is a Python file that fetches all the configuration and environment variables from respective files and stores them for use in the next phases. Then, based on the configuration variable, all the required operations are performed by calling functions from benchmarking and analysis units.

3.2. Benchmarking Unit

This unit has a main.py file, which contains all the Python operations required to perform the benchmarking, including functions to connect to the database and obtain and save the benchmarking results. In addition, system information such as RAM, processor, database version, CPU threads, etc., is saved in a JSON file for future analysis. All these functions are used by the stage.py file for benchmarking.

3.3. Analysis Unit

This unit contains all the components required to perform an analysis of the results. The analysis.py file fetches all the results to perform different statistical and graphical analyses as well as compare the results from both databases.

3.4. Comparative Analysis and Benchmarking Process Flow

The activity diagram defines the dynamic behavior of the modeled system and assists in understanding program flow at a high level. Figure 4 shows the behavior of the database benchmarking framework.

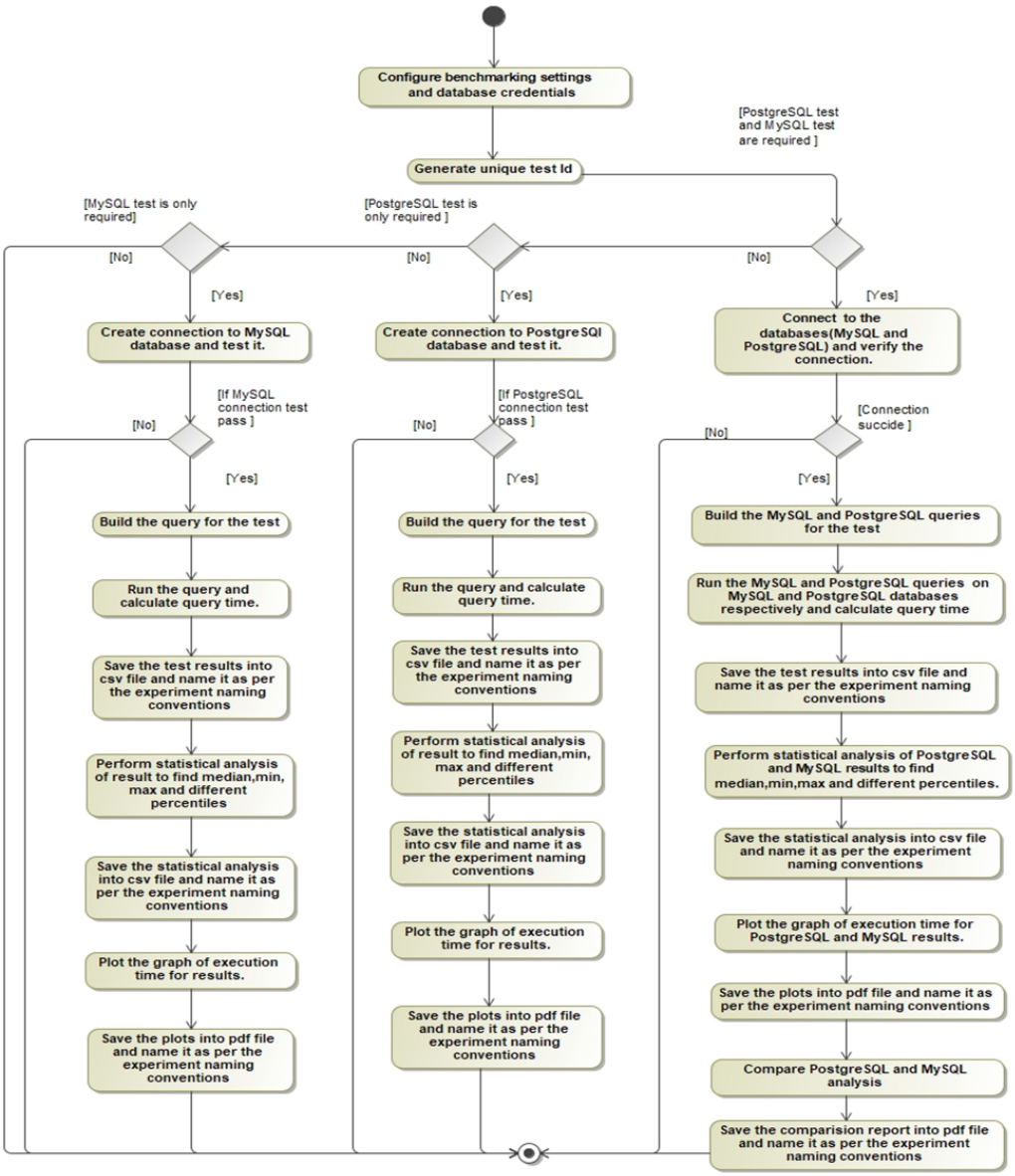

Figure 4. Database benchmarking activity diagram.

Initially, configuration variables are set based on the nature of the benchmarking test. Along with configuration variables, environment variables are required for the database. Once this information is provided, it is fetched and stored until the end of the test and used at different stages. Next, it creates a unique test ID to be used to identify the test.

Later, the configuration variables are scanned to determine if both PostgreSQL and MySQL tests are required or not. If they are required, then the following steps must be followed to run the test:

- First, a database connection is made and verified; failure to connect to the database would terminate the test.
- If the connections succeed for both databases, the configuration variables are used to build MySQL and PostgreSQL queries for benchmarking.
- Queries are executed on the respective databases.

The number of times a query should be executed depends on the configuration variables 'runs' and 'test_cycles'. For example, if 'runs' is 100 and 'test_cycles' is 5, then each query will be executed 100 times across 5 cycles (a total of 500 executions). Adding cycles helps determine performance patterns after executing queries. Simultaneously, the following steps are followed:

- Query time/latency (time between query trigger and completion) is calculated.
- Results are saved in CSV files named according to experimental file naming conventions (explained in Section 3.3).

These results undergo various analyses:

1. A statistical analysis is conducted to find the median, min, max, and percentiles
2. A graphical analysis is conducted to plot query time vs. runs for the complete test/cycles.

The analyses are saved in JSON (statistical) and PDF (graphical) files per the naming conventions. These analyses help identify patterns in the MySQL/PostgreSQL results; comparing both reveals overall patterns throughout tests. The subsequent steps include creating comparison PDFs showing the MySQL/PostgreSQL query times/moving averages for smoother patterns before completing the tests. If neither PostgreSQL nor MySQL tests are required, we check if a PostgreSQL test is needed:

- Create/test connection; failure terminates test.
- Build/execute queries using configuration variables; calculate/save query time/results in CSV.

The results undergo statistical/graphical analyses and are saved as JSON/PDF files, marking the end of the test. Similarly, MySQL tests are conducted if needed; otherwise, we terminate the test if neither is required.

3.5. Experimental File Naming Conventions

Result file: The query time/latency results are saved in a CSV file with a name determined according to the experimental file naming conventions, i.e., database name_table name_query type_number of runs_test Id.

For example, the name "postgresql_users_select_r_100_t_1666812060_1.csv" explains that the database PostgreSQL was tested for the user table by running a select query 100 times. The format shown in Table 1 below is used to save the test results.

Table 1. Result format.

Column Name	Date	Table	Query	Execution Time (s)	Cycle No
Details/example	Unix time stamp	users	Select * from users;	0.005880188000446651	1

Statistical and graphical analysis files: The result data are used to find statistical information like the median, max, min, etc. This information is then saved as a JSON file with a name such as "analysis_postgresql_users_select_r_100_t_1666812060_1.json", which has a similar meaning to the result file, except that it has the word "analysis" at the beginning to indicate that it is the analysis file. In the case of graphs, the plots are saved a a pdf file with a name such as "plots_postgresql_users_select_r_100_t_1666812060_1.pdf".

4. Experiments and Results

In continuous authentication, two important database operations are select and insert queries. The system needs to insert new records from the user and fetch records from the database for user authentication. Therefore, it is important to benchmark these operations under different conditions to identify the database with the lowest latency in a production environment. The data seed below is used for the experiment.

4.1. Data Seed for Experiment

For a faultless evaluation of databases in production-like cases, it is crucial to use a dataset similar to production data for benchmarking. A database table with the columns shown in Table 2 was created to replicate the production environment.

Table 2. Dataset details.

Field	Type
SessionID	bigint
timestamp	int
type	tinyint(1)
x	int
y	int
Event	int
userId	varchar(255)

The table has nine columns: seven of type int or similar, and two of the varchar and DateTime type. To create data similar to those obtained in production, an SQL procedure was developed to insert records based on requirements. For example, if the input to the procedure is 1000, it would add one thousand new records to the table.

4.2. Experimental Setup

To replicate the production scenario, select and insert queries are executed under different conditions. The experiments are divided into two categories: primary experiments and complex experiments. In primary experiments, simple production scenarios are evaluated, whereas in complex experiments, more intricate scenarios are assessed.

4.2.1. Primary Experiments

In production, a query is not executed only once but multiple times. Therefore, in these experiments, queries are executed multiple times for large datasets to analyze query latency. For primary experiments, select and insert queries are assessed individually. Table 3 provides details of the experimental conditions, number of queries executed, and total records in the table.

Table 3. Primary experiment details.

Experimental Condition	Number of Times Query Executed (Runs)	Number of Records in Table
Select query to fetch all records from table	100	1 million
Select query with condition to fetch record for one user only	100	1 million
Insert new records in table	100	1 million

4.2.2. Complex Experiments

In production, databases do not perform only one operation like select or insert, but handle multiple operations simultaneously, which can degrade their performance.

It is critical to analyze each database under certain conditions while executing multiple operations simultaneously. For instance, in continuous authentication, the database must insert new records while responding to fetch requests. To study the database under these conditions, the experiments described in Table 4 were performed.

Table 4. Complex experiment details.

Experimental Condition	Number of Times Query Executed (Runs)	Number of Records in Table
Select query to fetch all records from table evaluated while database performs insert operations simultaneously	100	1 million
Select query with condition to fetch record for one user only while database performs insert operation parallelly	100	1 million
Insert new records in table while database performs select operation simultaneously	100	1 million

In complex experiments, select and insert queries are evaluated while the database performs other operations simultaneously. For example, in the first experiment, the data latency for one hundred select queries was calculated while the database executed insert operations simultaneously.

4.2.3. Hardware/Software Details

Table 5 shows the system configuration on which all experiments are performed.

Table 5. Detailed System Configuration.

Database Type	MySQL and PostgreSQL
Database Kind	SQL
Database Version MySQL	8.0.20
Database Version PostgreSQL	PostgreSQL 14.6
Operating System	Windows-10-10.0.19041-SP0
System Memory	11.650901794433594
CPU Type	Intel64 Family 6 Model 140 Stepping 1, Genuine Intel
Total Cores	8
Total Threads	1

4.3. Results

A total of six different experiments (Tables 3 and 4) were performed to evaluate the databases in various production-like scenarios. The results of both the primary and complex experiments are discussed below.

4.3.1. Primary Experiments Results

1. Select Query to Fetch All Records from the Table: A select query was executed 100 times to fetch all records from both the PostgreSQL and MySQL databases. Figure 5 shows that the execution time in MySQL varies between 9 ms and 12 ms, whereas in PostgreSQL (Figure 6), it ranges from 0.6ms to 0.8 ms—significantly lower than MySQL. Figure 7 compares the execution times for retrieving 1 million records from PostgreSQL and MySQL, clearly showing PostgreSQL's superior performance. Table 6 shows the statistics for the execution times of both databases, including the median, maximum, minimum, and percentile values. These statistics assist in evaluating database performance. In this experiment, PostgreSQL outperforms MySQL in all select query stats by a factor of 13.

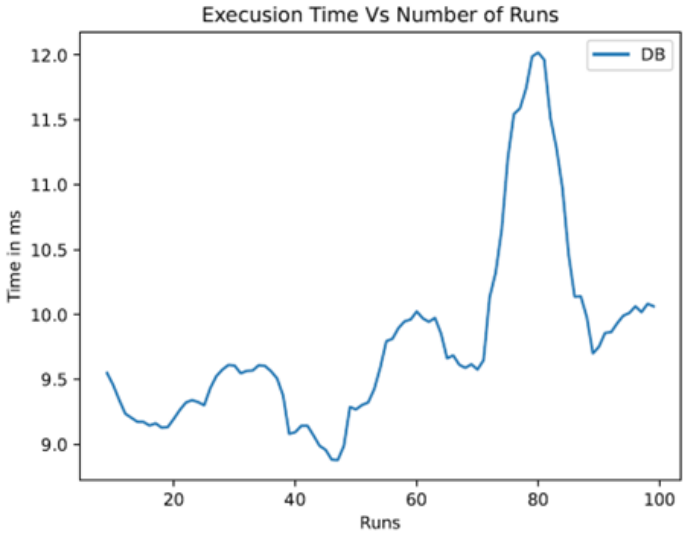

Figure 5. Select query execution time of MySQL for primary experiment one.

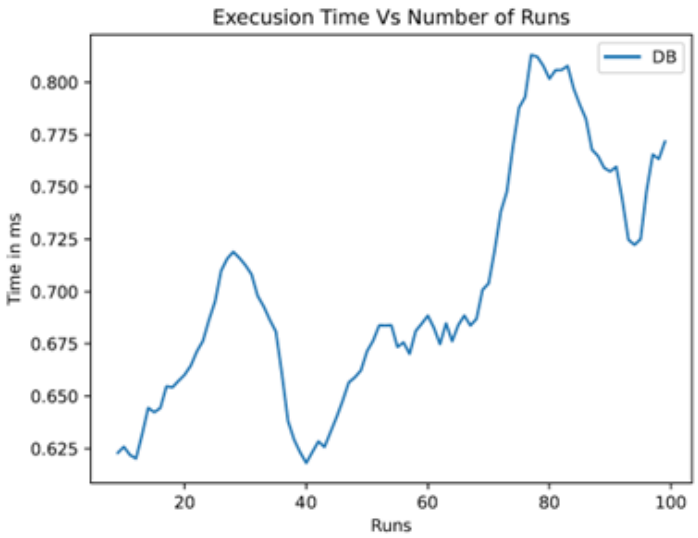

Figure 6. Select query execution time of PostgreSQL for primary experiment one.

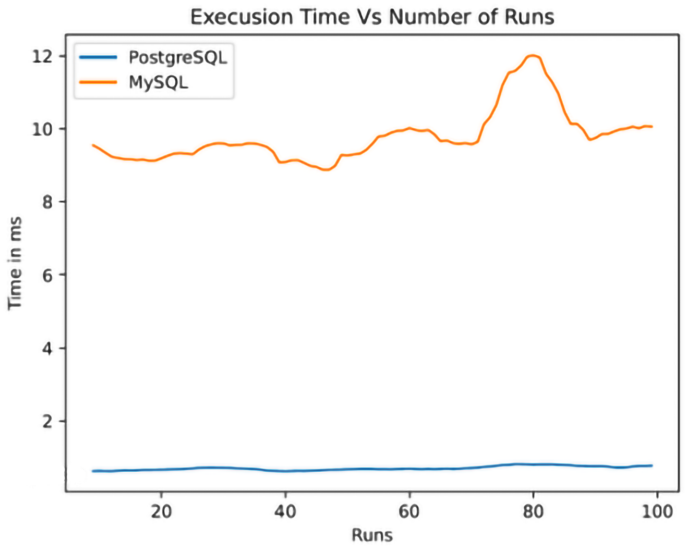

Figure 7. Select query comparison of MySQL and PostgreSQL for primary experiment one.

2. Select Query with Condition for One User: The second experiment tested a select statement with a where clause necessary for fetching data for specific users based on criteria. Using a query like select * from data where uname = 'clair', around ten thousand records were fetched from 1 million total records for the username 'clair'. MySQL took between 0.9 ms and 1ms, Figure 8, while PostgreSQL required about 0.09 ms to 0.13 ms, Figure 9—again outperforming MySQL. Figure 10 compares the results for select queries with where clauses between MySQL and PostgreSQL; PostgreSQL consistently outperforms MySQL. Table 7 shows that PostgreSQL beats MySQL at every stage, with a median execution time of 0.0726 ms versus MySQL's 0.8428 ms.

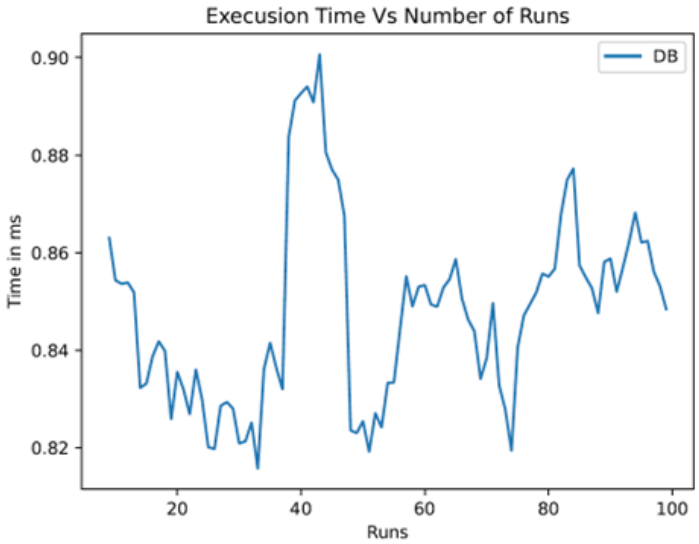

Figure 8. Select operation with where condition query execution time of MySQL for primary experiment two.

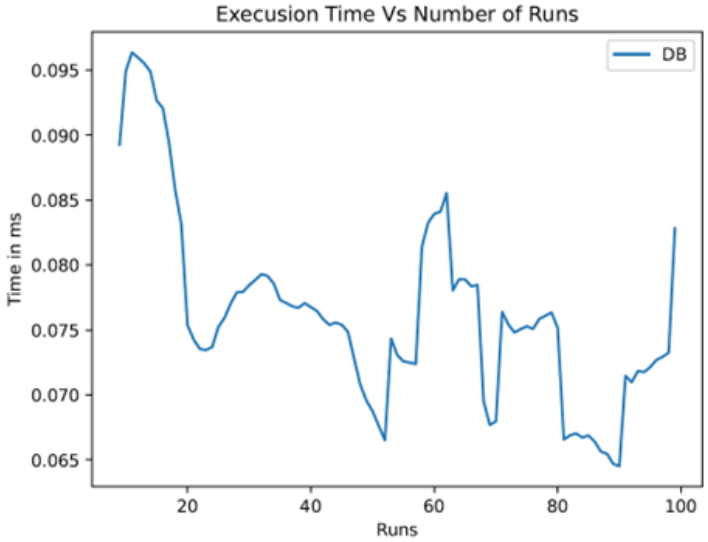

Figure 9. Select operation with where condition query execution time of PostgreSQL for primary experiment two.

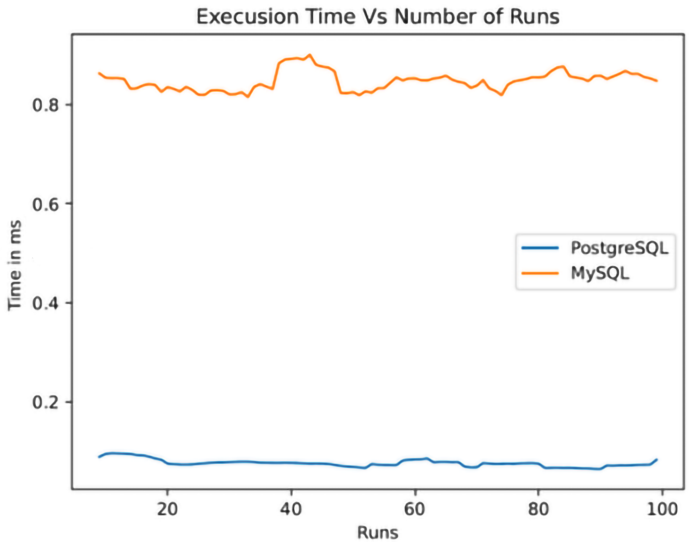

Figure 10. Select operation with where condition query comparison of MySQL and PostgreSQL for primary experiment two.

Table 6. Statistics for MySQL and PostgreSQL for primary experiment one.

Execution Time Stats	MySQL (ms)	PostgreSQL (ms)
Median	9.610416149999999	0.6922907000000014
Max	14.653671499999971	0.9570205999999928
Min	6.749867999999992	0.4895766000000003
Percentile 25%	9.338115	0.644354
Percentile 50%	9.610416	0.692291
Percentile 75%	9.962929	0.743065

Table 7. Statistical comparison for primary experiment two.

Execution Time Stats	MySQL (ms)	PostgreSQL (ms)
Median	0.8438375000000011	0.07268409999999956
Max	1.3509363000000008	0.1564174999999998
Min	0.6974038000000036	0.0596166
Percentile 25%	0.80234	0.066272
Percentile 50%	0.843838	0.072684
Percentile 75%	0.869998	0.078394

3. Insert New Records: For continuous authentication scenarios where large amounts of data are generated every second, an experiment was conducted with an insert query executed 100 times on both databases. Figures 11 and 12 show similar performance between MySQL and PostgreSQL, with execution times ranging from 0.0010 ms to 0.0030 ms for MySQL and from 0.0007 ms to 0.0014 ms for PostgreSQL. Figure 13 illustrates execution time graphs for both databases, showing comparable performance, with PostgreSQL slightly better than MySQL. The statistics from Table 8 confirm overlapping performance between MySQL and PostgreSQL, with a small differences in their statistical values.

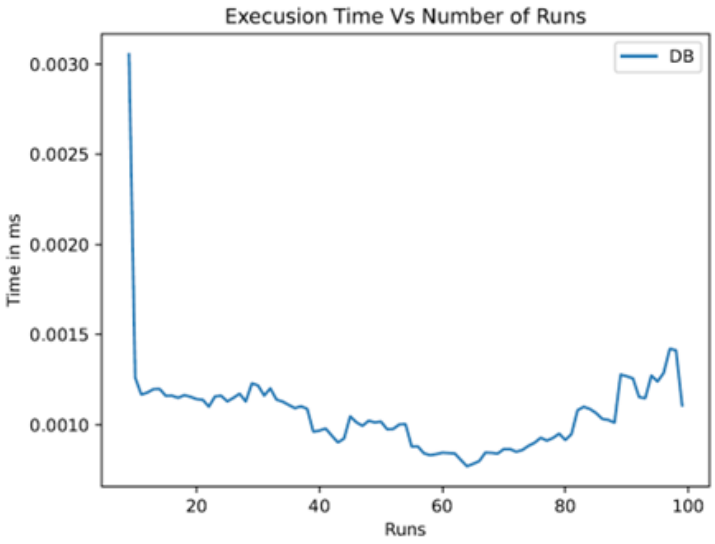

Figure 11. Insert query execution time of MySQL for primary experiment three.

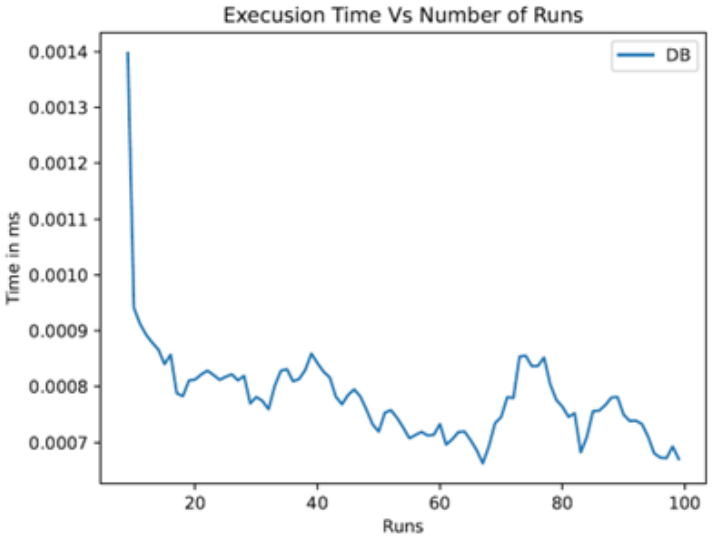

Figure 12. Insert query execution time of PostgreSQL for primary experiment three.

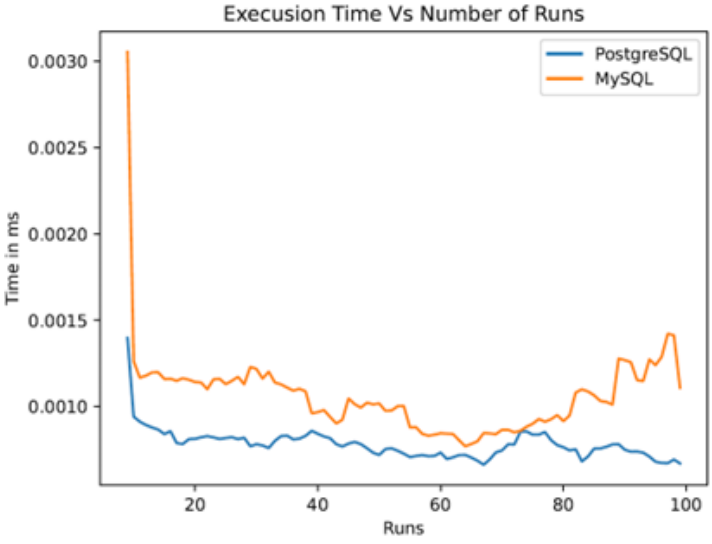

Figure 13. Insert query comparison of MySQL and PostgreSQL for primary experiment three.

Table 8. Statistical comparison for primary experiment three.

Execution Time Stats	MySQL (ms)	PostgreSQL (ms)
Median	0.0002600000000001	0.00012070000000005
Max	0.0030774	0.0005792
Min	0.0001705999999999	$8.740000000018178 \times 10^{-5}$
Percentile 25%	0.000224	0.000107
Percentile 50%	0.00026	0.000121
Percentile 75%	0.000305	0.000146

4.3.2. Complex Experimental Results

In production environments requiring handling multiple connections and requests simultaneously, the following steps must be taken:

1. Select Query While Performing Insert Operations: In continuous authentication scenarios requiring simultaneous insert and select operations every few seconds, this scenario evaluates select operation performance when inserts occur simultaneously. Figure 14 shows MySQL's select operation performance while executing simultaneous inserts; the execution time varies between 7 ms and 13 ms compared to PostgreSQL's range of 0.7 ms to 0.9 ms as shown in Figure 15. Figure 16 and Table 9 compare MySQL's degraded performance due to increasing record numbers against PostgreSQL's stable performance despite changes.

Table 9. Statistical comparison for complex experiment one.

Execution Time Stats	MySQL (ms)	PostgreSQL (ms)
Median	12.228753049999938	0.8172035000000051
Max	13.367786599999988	1.0093484000000004
Min	6.454262799999924	0.7369057999999953
Percentile 25%	8.864012	0.782431
Percentile 50%	12.228753	0.817204
Percentile 75%	12.642323	0.858551

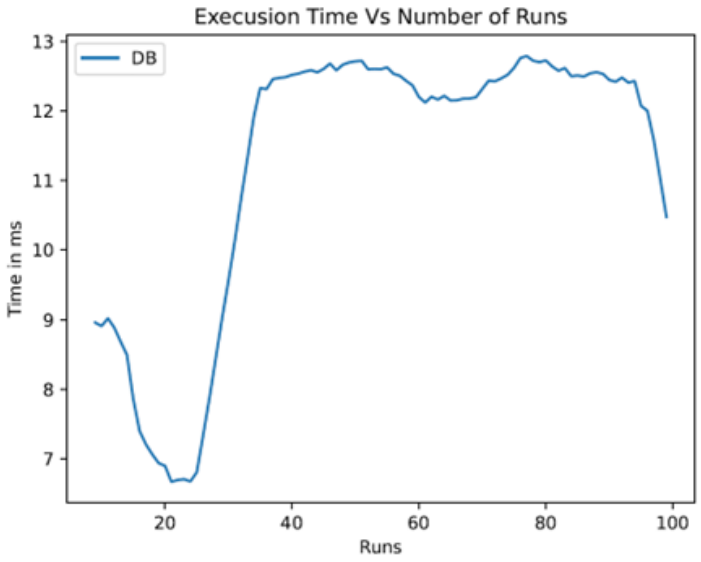

Figure 14. Select query execution time of MySQL with insert operation in parallel.

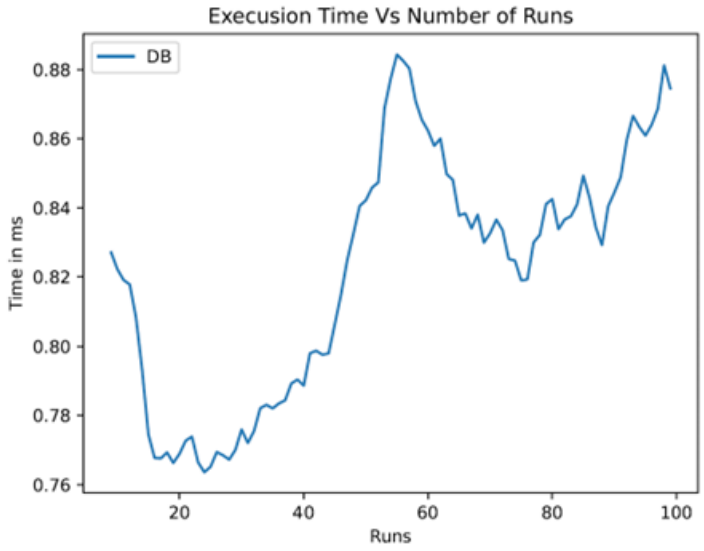

Figure 15. Select query execution time of PostgreSQL with insert operation in parallel.

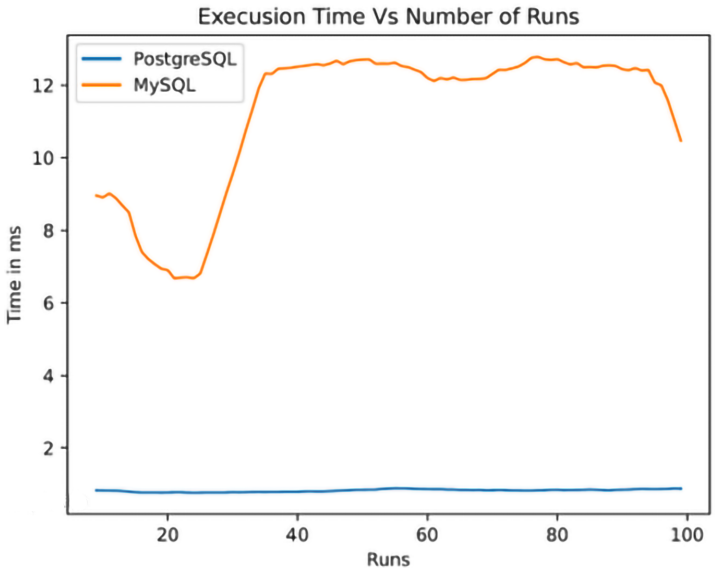

Figure 16. Select query comparison of MySQL and PostgreSQL with insert operation in parallel.

2. Select Query with Condition While Performing Insert Operations: This case evaluates a select operation with a where condition while executing parallel insert queries without adding additional records for the username 'clair'. Figures 17–19 show PostgreSQL outperforming MySQL again, with execution times ranging from 1 ms to 1.5 ms for MySQL compared to PostgreSQL's range of 0.07 ms to 0.13 ms. Table 10 displays statistics confirming PostgreSQL's superior performance by roughly nine times over MySQL.

Table 10. Statistical comparison for complex experiment two.

Execution Time Stats	MySQL (ms)	PostgreSQL (ms)
Median	1.253879149999996	0.0929318000000001
Max	1.7836954999999932	0.1886225999999999
Min	0.7546789999999994	0.0586152000000002
Percentile 25%	1.106501	0.086717
Percentile 50%	1.253879	0.092932
Percentile 75%	1.439175	0.10246

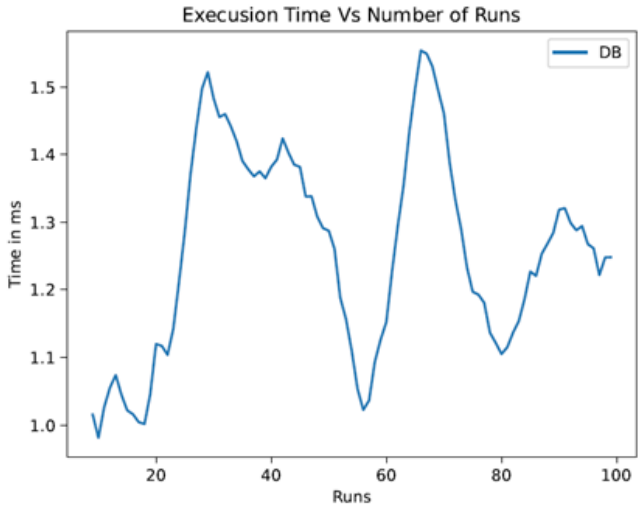

Figure 17. Select operation with where query execution time of MySQL with insert operation in parallel.

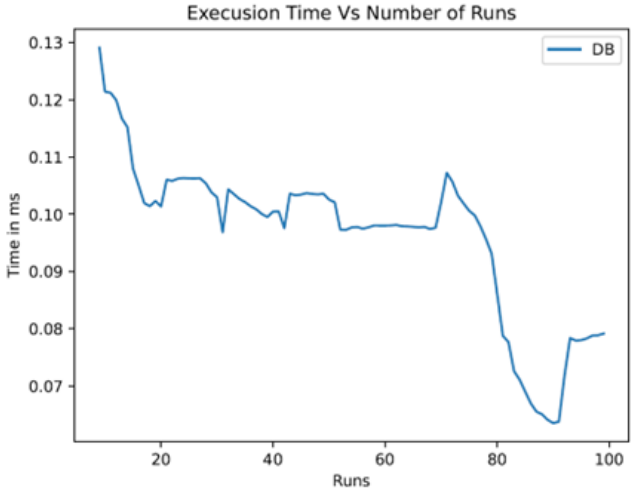

Figure 18. Select operation with where query execution time of PostgreSQL with insert operation in parallel.

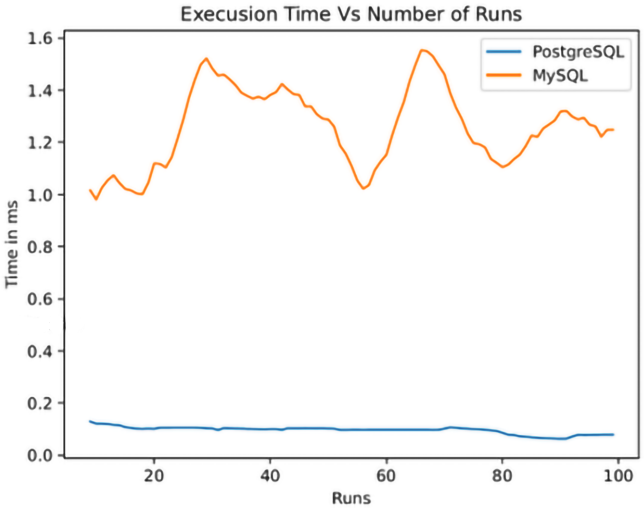

Figure 19. Select operation with where query comparison of MySQL and PostgreSQL with insert operation in parallel.

3. Insert New Records While Performing Select Operations: This scenario assesses insert operation performance during simultaneous select query execution. Figures 20 and 21 show performances where MySQL varies between 0.00020 ms and 0.00043 ms compared with PostgreSQL's range of 0.00010 ms to 0.00017 ms—demonstrating better performance by PostgreSQL during concurrent operations. Figure 22 compares the execution time results, showing clear superiority of PostgreSQL over MySQL, as confirmed by Table 11, showing median values of MySQL at 0.00020 ms versus 0.00010 ms for PostgreSQL.

Table 11. Statistical comparison for complex experiment three.

Execution Time Stats	MySQL (ms)	PostgreSQL (ms)
Median	0.00020265000000005	0.00010899999999995001
Max	0.001465	0.0005941999999999
Min	0.0001616000000002	$9.090000000000488 \times 10^{-5}$
Percentile 25%	0.000187	9.9×10^{-5}
Percentile 50%	0.000203	0.000109
Percentile 75%	0.000238	0.000121

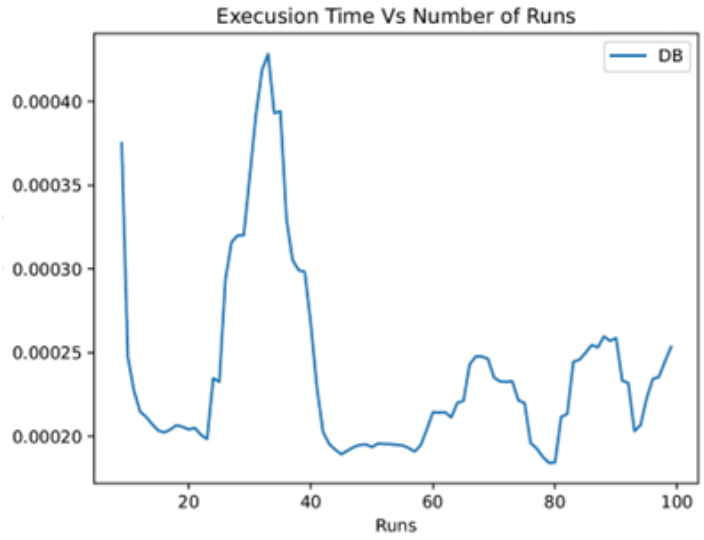

Figure 20. Insert query execution time of MySQL with select operation in parallel.

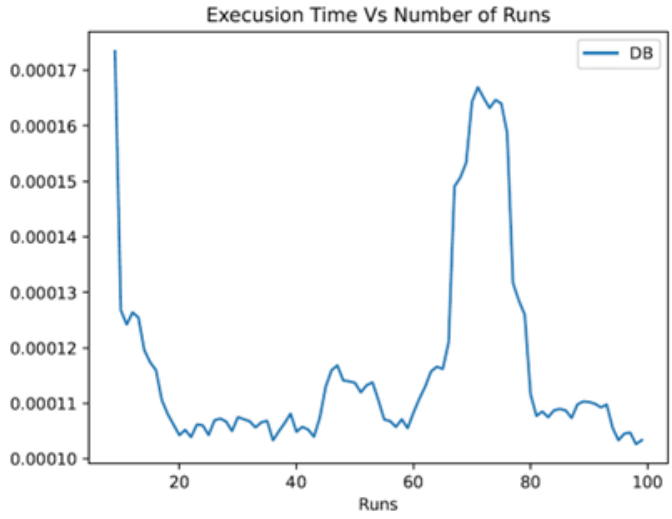

Figure 21. Insert query execution time of PostgreSQL with select operation in parallel.

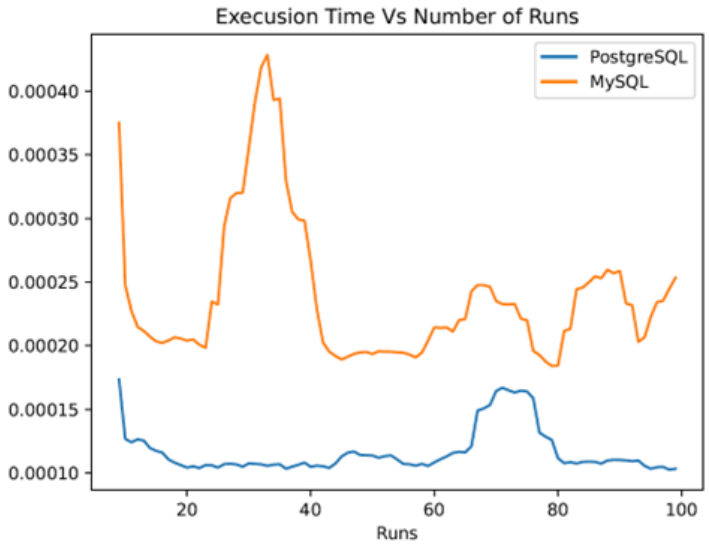

Figure 22. Insert query comparison of MySQL and PostgreSQL with select operation in parallel.

Also noteworthy is that these results can be attributed partly to PostgreSQL's robust support hierarchical data models, such as the adjacency list model, allowing efficient management insertion of complex tree-like data structures through its capability to handle recursive queries and optimize write-heavy operations, as evidenced by benchmark studies demonstrating superior write operation performance compared with MySQL [29].

5. Conclusions

This study provides valuable insights into the performance dynamics of two widely used relational database management systems (RDBMS), PostgreSQL and MySQL, under production-like scenarios typical of continuous user authentication systems.

5.1. Theoretical Implications

- Database Performance Understanding: This study enhances the theoretical framework for understanding database performance in high-demand environments. It provides a detailed comparison between PostgreSQL and MySQL, emphasizing the importance of low latency in continuous user authentication systems.
- Benchmarking Methodology: The development of a Python-based benchmarking framework contributes to the theoretical tools available for database performance evaluation, offering a reusable and adaptable method for other researchers and practitioners.

5.2. Practical Implications

- Database Selection: The findings of this study suggest that PostgreSQL is more suitable for continuous authentication systems due to its superior performance in handling select operations both with and without conditions. This can guide organizations in choosing the right database system to optimize their application performance.
- Benchmarking Framework: The development of a flexible benchmarking framework allows database administrators and developers to assess different databases' performance efficiently. This framework can be customized for various operational needs, facilitating better decision-making regarding database deployment.
- Performance Optimization: Understanding the specific strengths and weaknesses of PostgreSQL and MySQL in different operational scenarios can help optimize query execution times and overall system responsiveness.

5.3. Limitations

- Scope of Databases: This study focuses only on PostgreSQL and MySQL, which limits its applicability to other types of databases, such as NoSQL [30] or NewSQL systems [31], which might perform differently under similar conditions.
- Simulated Environment: Although these experiments simulate production-like scenarios, they may not capture all real-world variables, such as network latency or concurrent user interactions, that could affect database performance.
- Complexity of Operations: This study primarily evaluates basic operations like select and insert queries. More complex queries or transactions involving multiple tables and joins could yield different results.
- Hardware Constraints: Our performance results are contingent on the specific hardware and software configurations used during testing. Different setups might produce varying outcomes.
- Relating to Relevant Works: Relating this study to the existing literature is essential to underscore its significance and contribution. However, this task presents challenges due to this study's specific focus on continuous user authentication needs. This niche area means that while there is a substantial body of work on database benchmarking in general, studies specifically addressing the unique demands of continuous authentication systems are limited.

5.4. Future Research Directions

In the future, to improve benchmarking, it is crucial to focus on cloud-native and containerized environmental benchmarks, as there is growing adoption of cloud-based solutions like Google Cloud SQL [32], Azure databases for PostgreSQL/MySQL [33], and Amazon RDS [34]. Therefore, it would be valuable to create a benchmarking tool to evaluate performance under dynamic scaling, integration with cloud-native services, and network latency. Moreover, containers like docker are becoming more popular for database deployment, and benchmarking PostgreSQL and MySQL in containerized environments could focus on performance with tools like Kubernetes [35], assessing resource allocation efficiency, behavior under container restart or failure, and database performance while scaling. These tailored benchmarks would provide a more accurate evaluation of the real-world performance of databases across modern deployment platforms.

Author Contributions: Conceptualization, S.V.S. and A.O.; methodology, S.V.S. and A.O.; software, S.V.S.; validation, S.V.S. and A.O.; formal analysis, S.V.S. and A.O.; investigation, S.V.S. and A.O.; resources, S.V.S. and A.O.; data curation, S.V.S. and A.O.; writing—original draft, S.V.S.; writing—review and editing, S.V.S. and A.O.; project administration, A.O.; funding acquisition, A.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Salunke, S.; Ouda, A.; Gagne, J. Transfer Learning for Behavioral Biometrics-based Continuous User Authentication. In Proceedings of the 2022 International Symposium on Networks, Computers and Communications (ISNCC), Shenzhen, China, 19–22 July 2022; pp. 1–6.
2. Salunke, S.V.; Ouda, A. Ensemble Learning to Enhance Continuous User Authentication For Real World Environments. In Proceedings of the 2023 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom), IEEE, Istanbul, Turkiye, 4–7 July 2023; pp. 102–108.
3. Yao, S.B.; Hevner, A.R.; Young-Myers, H. Analysis of database system architectures using benchmarks. *IEEE Trans. Softw. Eng.* **1987**, *SE-13*, 709–725. [CrossRef]
4. Almeida, D. Performance Comparison of Redis, Memcached, MySQL, and PostgreSQL: A Study on Key-Value and Relational Databases. In Proceedings of the 2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon), Singapore, 18–19 August 2023; pp. 902–907.

5. Han, R.; John, L.K.; Zhan, J. Benchmarking Big Data Systems: A Review. *IEEE Trans. Serv. Comput.* **2018**, *11*, 580–597. [CrossRef]
6. Weng, S.; Wang, Q.; Qu, L.; Zhang, R.; Cai, P.; Qian, W.; Zhou, A. Lauca: A Workload Duplicator for Benchmarking Transactional Database Performance. *IEEE Trans. Knowl. Data Eng.* **2024**, *36*, 3180–3194. [CrossRef]
7. Zhou, G.; Huang, L.; Li, Z.; Tian, H.; Zhang, B.; Fu, M.; Feng, Y.; Huang, C. Intever Public Database for Arcing Event Detection: Feature Analysis, Benchmark Test, and Multi-Scale CNN Application. *IEEE Trans. Instrum. Meas.* **2021**, *70*, 3518515. [CrossRef]
8. Ciolli, G.; Mejías, B.; Angelakos, J.; Kumar, V.; Riggs, S. *PostgreSQL 16 Administration Cookbook: Solve Real-World Database Administration Challenges with 180+ Practical Recipes and Best Practices*; Packt Publishing Ltd.: Birmingham, UK, 2023.
9. Murach, J. *Murach's MySQL*, 3rd ed.; Mike Murach Associates: Fresno, CA, USA, 2019.
10. Edrah, A.; Ouda, A. Enhanced Security Access Control Using Statistical-Based Legitimate or Counterfeit Identification System. *Computers* **2024**, *13*, 159. [CrossRef]
11. Aref, Y.; Ouda, A. HSM4SSL: Leveraging HSMs for Enhanced Intra-Domain Security. *Future Internet* **2024**, *16*, 148. [CrossRef]
12. Ibrahim, A.; Ouda, A. Innovative data authentication model. In Proceedings of the 2016 IEEE 7th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 13–15 October 2016; pp. 1–7.
13. Bansal, P.; Ouda, A. Study on integration of fastapi and machine learning for continuous authentication of behavioral biometrics. In Proceedings of the 2022 International Symposium on Networks, Computers and Communications (ISNCC), Shenzhen, China, 19–22 July 2022; pp. 1–6.
14. Stonebraker, M.; Rowe, L.A.; Hirohama, M. The implementation of POSTGRES. *IEEE Trans. Knowl. Data Eng.* **1990**, *2*, 125–142. [CrossRef] [PubMed]
15. A Brief History of PostgreSQL. In *PostgreSQL Documentation*; The PostgreSQL Global Development Group: Athens, Greece, 2018.
16. Bernstein, P.A.; Goodman, N. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst. (TODS)* **1983**, *8*, 465–483. [CrossRef]
17. Haerder, T.; Reuter, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv. (CSUR)* **1983**, *15*, 287–317. [CrossRef]
18. Comer, D. Ubiquitous B-tree. *ACM Comput. Surv. (CSUR)* **1979**, *11*, 121–137. [CrossRef]
19. Hellerstein, J.M.; Naughton, J.F.; Pfeffer, A. *Generalized Search Trees for Database Systems*; University of Wisconsin-Madison Department of Computer Sciences: Madison, WI, USA, 1995.
20. Bartunov, O.; Sigaev, T. Full-Text Search in Postgresql. PostgreSQL Documentation 2024. Available online: https://www.postgresql.org/docs/current/textsearch.html (accessed on 17 October 2024).
21. Ramsey, P. Postgres Indexing: When Does BRIN Win? Crunchy Data. 2022. Available online: https://www.crunchydata.com/blog/postgres-indexing-when-does-brin-win (accessed on 13 October 2024).
22. Heck, W. Using MySQL in your organisation. In Proceedings of the INTED2009 Proceedings, IATED, Valencia, Spain, 9–11 March 2009; pp. 3686–3694.
23. Seghier, N.B.; Kazar, O. Performance benchmarking and comparison of NoSQL databases: Redis vs mongodb vs Cassandra using YCSB tool. In Proceedings of the 2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI), Tebessa, Algeria, 21–22 September 2021; pp. 1–6.
24. Filip, P.; Čegan, L. Comparison of MySQL and MongoDB with focus on performance. In Proceedings of the 2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), Jakarta, Indonesia, 19–20 November 2020; pp. 184–187.
25. Tongkaw, S.; Tongkaw, A. A comparison of database performance of MariaDB and MySQL with OLTP workload. In Proceedings of the 2016 IEEE Conference on Open Systems (ICOS), Langkawi, Malaysia, 10–12 October 2016; pp. 17–119.
26. Kenler, E.; Razzoli, F. *MariaDB Essentials*; Packt Publishing Ltd.: Birmingham, UK, 2015.
27. Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. Xen and the art of virtualization. *ACM SIGOPS Oper. Syst. Rev.* **2003**, *37*, 164–177. [CrossRef]
28. Bog, A.; Kruger, J.; Schaffner, J. A composite benchmark for online transaction processing and operational reporting. In Proceedings of the 2008 IEEE Symposium on Advanced Management of Information for Globalized Enterprises (AMIGE), Tianjin, China, 28–29 September 2008; pp. 1–5.
29. Callaghan, M. MySQL and Postgres vs the Insert Benchmark on a Large Server. 2024. Available online: https://smalldatum.blogspot.com/2024/09/mysql-and-postgres-vs-insert-benchmark.html (accessed on 13 October 2024).
30. Han, J.; Haihong, E.; Le, G.; Du, J. Survey on NoSQL database. In Proceedings of the 2011 6th International Conference on Pervasive Computing and Applications, Port Elizabeth, South Africa, 26–28 October 2011; pp. 363–366.
31. Kaur, K.; Sachdeva, M. Performance evaluation of NewSQL databases. In Proceedings of the 2017 International Conference on Inventive Systems and Control (ICISC), Coimbatore, India, 19–20 January 2017; pp. 1–5.
32. Sabharwal, N.; Edward, S.G. Hands on Google Cloud SQL and Cloud Spanner. In *Deployment, Administration and Use Cases with Python*; Apress: New York, NY, USA, 2019.
33. Shaik, B.; Vallarapu, A. *Beginning PostgreSQL on the Cloud: Simplifying Database as a Service on Cloud Platforms*; Apress: New York, NY, USA, 2018.

34. Armenatzoglou, N.; Basu, S.; Bhanoori, N.; Cai, M.; Chainani, N.; Chinta, K.; Govindaraju, V.; Green, T.J.; Gupta, M.; Hillig, S.; et al. Amazon Redshift re-invented. In Proceedings of the 2022 International Conference on Management of Data, Paphos, Cyprus, 12–17 June 2022; pp. 2205–2217.
35. Kubernetes, T. Kubernetes. *Kubernetes. Retrieved May* **2019**, *24*, 2019.