Information Research Communications, 2025; 2(1):98-119.
https://inforescom.org

Research Article

Comparative Analysis of Relational and Non-Relational Database Models: A Case Study on Travel Booking Systems

Chan Pui Ying[1], Joselyn Chin Shi Min[1], Gan Yi Jean[1], Leong Zheng Xuan[1], Nurjiha Natasha Binti Md Rafi[1], Sathishkumar Veerappampalayam Easwaramoorthy[1,*], Usha Moorthy[2]

[1]School of Computing and Artificial Intelligence, Faculty of Engineering and Technology, Sunway University, No. 5, Jalan Universiti, Bandar Sunway, Selangor Darul Ehsan, MALAYSIA.
[2]Department of Information Technology, Manipal Institute of Technology Bengaluru, Manipal Academy of Higher Education, Manipal, Karnataka, INDIA.

ABSTRACT

Aim/Background: The study aims to evaluate and compare relational and non-relational (NoSQL) database models in the context of travel booking systems. Traditional relational databases like MySQL are known for their data integrity and structured schema, while NoSQL models like ArangoDB, Apache Cassandra, Memcached, and MongoDB offer scalability and flexibility. This research investigates which model performs best across various database operations relevant to real-world scenarios. **Methodology:** The research used both theoretical and practical methods. A literature review was conducted using sources such as Google Scholar, IEEE Xplore, and database documentation. The team implemented a travel booking system to perform standardized operations like insert, update, delete, retrieve, access control, and integrity checks using five selected DBMSs. Performance metrics included execution time, CPU usage, throughput, and constraint handling, and were analyzed using visualizations and comparative tables. **Results:** Apache Cassandra showed the best overall performance for flight booking systems due to high throughput and scalability. MySQL performed exceptionally in data integrity and consistent performance under constraints. ArangoDB showed flexibility and low insertion time but required a learning curve. Memcached excelled in insert/delete throughput but lacked persistence. MongoDB offered schema flexibility but lagged in constrained scenarios and multi-document transactions. **Discussion:** The findings reveal that database selection should align with specific application needs. NoSQL models are preferable for high-speed, flexible operations, while relational databases are superior for structured, integrity-focused use cases. The study corroborates prior research advocating NoSQL for big data applications, yet highlights the enduring value of RDBMSs in mission-critical systems. **Conclusion:** Apache Cassandra is the most appropriate choice for large-scale, dynamic systems like travel booking due to its high availability and performance. MySQL and ArangoDB also have strong points for specific tasks. The research emphasizes that no one-size-fits-all model exists, and DBMS selection must consider operation type, data size, and performance requirements.

Keywords: Relational Databases, NoSQL Databases, Database Performance Evaluation, Travel Booking Systems, Data Integrity and Access Control, Scalability and Throughput.

Correspondence:
Dr. Sathishkumar Veerappampalayam Easwaramoorthy
School of Computing and Artificial Intelligence, Faculty of Engineering and Technology, Sunway University, No. 5, Jalan Universiti, Bandar Sunway, 47500 Selangor Darul Ehsan, MALAYSIA.
Email: sathishv@sunway.edu.my

Received: 19-02-2025;
Revised: 04-04-2025;
Accepted: 28-05-2025.

INTRODUCTION

Background of Study

Databases are one of the most important building blocks of any present-day information systems, as they allow for both collecting and organizing these resources. Originally, important data was stored through Relational Database Management Systems (RDBMS) such as MySQL, with Structured Query Language (SQL) to manage the relational database as per the schema. In context of RDBMS, the data is stored in the form of tables with rows and columns and one table can have reference to other table by using foreign keys and the join operation can be used to get relationship between one table and another table. This structured approach safeguards the data, and it reduces or eliminates option and makes RDBMS rather suitable for many traditional applications Seghier, N. B., & Kazar, O., (2021).

Nevertheless, with big data appearing on the scene, the data has become more diverse, fast, and massive, creating new difficulties for RDBMS. Use and demand for easy to scale and more performance-based databases make it easier to use Non-Relational or NoSQL databases. NoSQL is classified into

DOI: 10.5530/irc.2.1.7

several types each of which is optimized to match a type of data and application requirements more than relational. Chen, J.-K., & Lee, W.-Z., (2019).

Graph Model

ArangoDB and all the other graph databases arrange data in nodes and edges that stand for the entities and their connecting means, respectively. This model performs the best in cases when there is a need to define a connection between variables, and this connection is variable itself, like in social networks, recommendation systems, and fraud detection. Walke, D., *et al.*, (2023).

Wide-Column Model

Wide-column databases such as Apache Cassandra stores data in rows and/or columns and unlike the normal RDBMS data model it differs in that it can accommodate more flexible data models. The rows can be defined independently from the columns and the table, which makes the structure perfect for huge, distributed datasets with differing schema. This model is particularly effective for time series data, real-time analysis and logging in large systems.

Key-Value Model

A type of NoSQL databases is key-value databases where information is stored in form of a key-value type of pairs as is seen in Memcached. This basic model delivers high throughput read and write capabilities that are ideal for caching, user sessions and real-time processing when initial access to data is paramount.

Document Model

Document store databases such as MongoDB really store data in JSON like documents with possibility to nest and contain data of different types. This model is flexible in representation since it enables usage of XML structure in flexible content management systems, e-commerce and any application that deals with dynamic and hierarchical data Zaniewicz, N., & Salamończyk, A., (2022).

Problem Statement

Although today more and more systems use NoSQL databases, most organizations still use relational databases intensively, which complicates the decision on the choice of the appropriate database model for certain applications. Other issues making this decision hard are the absence of a definite protocol on the distinctions between relational and non-relational databases and concrete comparisons of the two in the creation, operation, querying, security, and integrity of data. As such, this research seeks to fill this research gap by developing a comparative study between a Relational Database Management System (MySQL) and NoSQL Database Management Systems of ArangoDB, Apache Cassandra, Memcached, and MongoDB.

Significance of Study

The research is valuable for the database administrators, system architects, and IT decision makers who are always under pressure of choosing the most appropriate database model relevant to their needs. Through presenting the detailed findings of the comparison between Relational Database Model (MySQL) and other NoSQL databases, this research will be beneficiary in terms of sharing the strengths and weaknesses of such models. Such distinctions will help organizations decide how to most efficiently meet the requirements posed by current data applications and the nature of data itself. In addition, this study will help to enrich the scientific discourse on database technologies, thus contributing to a more effective understanding of real-life use of both relational and non-relational database solutions along with their performance profiles.

Research Questions

What are the differences between relational and non-relational database models in terms of data creation, data manipulation, data retrieval, access control, and data integrity?

Which database model is the most suitable for travel booking system scenario based on these aspects?

To answer these research questions, the study will adopt the use of the comparison of these database models in the scenario of the travel booking system. This will entail determining each database model's efficiency in use scenarios, a real-world approach to assessing their applicability in managing flight booking data.

LITERATURE REVIEW

Data Creation

The process of creating data is a significant step in the management of a database and includes the following steps. First of all, creating a data model which is a kind of a plan describing the structure of the database and how the data is stored and connected meeting users' needs. It also contains the definition of the tables and, field in the tables and their types collectively their definitions and the relation between the tables Daugirdas, D., & Zatorskis, J., (2023). When the schema is developed then, tables are formed using such SQL commands such as create table comprising of fields and data types. When creating tables, it is necessary to set up the primary and foreign keys through which the data integrity is imposed and necessary links between related data are made Sug, H., (2020).

After the tables have been created in the database a load data process takes place. This can be accomplished by writing simple INSERT SQL statements to insert records on an exclusive basis or, using bulk insert operations that are characteristic of Extract, Transform, Load (ETL) tools Wang, X., *et al.*, (2022). ETL tools simplify the data loading process through retrieval of data from different sources, the processing of data into a form that fits in the database schema and then loading the data into the tables. For instance, it could involve the data collected from CSV files or APIs and where data is pre-cleaned and formatted to match database specifications, the data transfer is done in large volumes. Such systematic approach to data creation is useful in guaranteeing the proper laying down of a well-structured and had populated database to ease data management and captures Bansal, S., & Kagemann, S., (2015).

Data Manipulation

Data manipulation involves the processes and methods of interaction with the data as well as with databases. Structured Query Language or SQL is the common language for these functions: SELECT, INSERT, UPDATE, DELETE. Data manipulation guarantees the credibility, coherence, and availability of data for the intended use. Retrieval is done by using the SELECT statement where data can be requested based on certain criteria in reference to SQL. The INSERT statement can be used to write new records to a database table, which enables the expansion of data. The UPDATE statement will allow editing records to make sure that data will always be up to date. The DELETE statement erases records from a database which helps in the management of the database and its space. Collectively, these operations facilitate the accuracy, uniformity, and usability of data for proper functioning of database systems. Data manipulation techniques are crucial in data management to ensure the accuracy of data, quick execution of queries, and performance of transactions, which are vital for the proper functioning of today's DBMS Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). Data manipulation plays a central role in the database management since it deals directly and has a direct impact with the quality, reliability, and efficiency of data-driven processes and applications. That is why it allows the manipulation of data in real-time, including data analysis, decision-making, and operations automation. It is significant to properly manipulate data to the extent that data continues to be an organizational asset through helping in intelligence gathering and business operations Elmasri, R., & Navathe, S., (2013).

Data Retrieval

Data retrieval in databases is one of the prime operations that involves extracting and presenting data stored within a database system in a significant manner. It is a process through which users can query and fetch the required data for several applications. The retrieval process starts with the formulation of a query that is a structured request to extract specific data based on defined criteria. The database management system then parses, optimises, and executes this query to fetch the requested data from the database. In databases, data accessing is also an important factor by which the performance of the database can be maintained, and it includes the concepts of indexes, query optimization techniques and few of them utilize caching mechanisms. Besides that, it is understood that newer databases offer simple questions, joined queries, aggregate functions, and full text searches. This versatility makes it possible for users to execute multiple operations right from simple data searches to complex data analysis, and even data reporting. Besides providing timely and rapid access to the required data, data retrieval helps in facilitating the decision-making process by supplying correct and pertinent data findings Callan, J., (2005).

Access Control

Another aspect of information security is access control which is defined as the ability to regulate and restrict an entity's access to a specific resource or object in a system. The main objective of access control is to safeguard data and the integrity of systems by permitting only those who are authorized to use the systems. The access control mechanisms are supposed to enforce the security policies and avoid any form of intrusion. They include several functions like user identification and authentication functions that check the identity of persons who want to access. Moreover, access control measures assist in preventing the data from being accessed or changed by unauthorized personnel, thus enhancing the security of information Samarati, P., & De Vimercati, S. C., (2001). Access control also has features for intrusion to ensure that the security of the system is not compromised. By applying the right access control measures, the risks are reduced, and the information is safeguarded from the wrong people and dangers. An example of the access control mechanisms is Access Control Lists (ACLs) that are used to limit the access to the data on the shared systems by connecting to the objects such as files and identifying the users or groups that are permitted to access the objects Petković, M., & Jonker, W., (2007). Therefore, access control is one of the most important aspects of information security that is vital in the protection of data confidentiality, data integrity, and data availability in any system.

Data Integrity

Data integrity is essential in a database, ensuring accurate, complete and consistent information at any stage of its use. It is

Table 1: Relational Model Structure.

Student ID	Name	Major	Birth Date	Enrollment Date
1	Samantha Brooks	Computer Science	2000-01-15	2023-09-01
2	John Robin	Business Management	1999-11-11	2023-09-01
3	Ava Johnson	Engineering	2001-03-1	2022-03-14
4	Justin Hayes	Finance	2002-09-9	2022-03-14

crucial to maintain data reliability, as data that has been corrupted or altered can lead to wrong conclusion and strategic mistakes. Another important aspect that differentiates data integrity from data security is that the former is concerned with the quality of the data, while the latter is concerned with the protection of the data from unauthorized access and breaches. Data integrity entails several processes such as accurate data collection, error checking and cybersecurity measures to prevent data breaches from internal or external threats Harvard Business School, (2021). Data integrity is not only important for decision making but also for maintaining trust and credibility with stakeholders by protecting sensitive information and adhering to legal standards. Furthermore, data integrity improves the operational performance since it reduces errors and the time and effort required to rectify them. Finally, it is crucial to emphasize that data integrity as a key to safeguarding the data, preserve reputation and guarantee the accuracy of organizational operations and results.

Table 2: **Basic SQL Commands.**

Operation	SQL Command
Select data	SELECT * FROM Students;
Insert data	INSERT INTO Students (Student ID, Name, Major) VALUES (1, 'Alice Johnson', 'Computer Sci');
Update data	UPDATE Students SET Major = 'Data Science' WHERE Student ID = 1;
Delete table	DELETE FROM Students WHERE Student ID = 1;
Create table	CREATE TABLE Students (Student ID INT PRIMARY KEY, Name VARCHAR (100), Major VARCHAR (100));

Relational Database Model (MySQL)

Edgar F. Codd proposed the relational database model in 1970 and that has remained the mainstream for decades of database management systems. MySQL, which is developed by Oracle Corporation, utilises a relational database model that is used to organise data into tables which are linked by relationships. It is known for its robustness, reliability, and performance, thus making it a popular choice for web applications, data warehousing, and business applications Blansit, B. D., (2006).

MySQL organises data into tables that consist of rows and columns. In this matter, each table represents a specific entity type whereas each row within a table corresponds to a unique instance of that entity that is identified by a primary key. Columns represent the attributes of the entity to produce a structured and consistent format for data storage. According to Table 1, The 'Students' table records detailed information about each student such as their ID, name, major, birthdate, and enrolment date.

As shown in Table 2, the SQL commands table provides examples of basic SQL operations for managing the 'Students' table consisting of selecting, inserting, updating, and deleting records.

However, while the relational model remains highly effective, especially for structured data and complex queries, it faces challenges with unstructured data and scalability in certain

Table 3: **Summary for Each DBMS.**

Database	Data Distribution	Schema	Constraints	Consistency
MySQL	Data is not inherently distributed across a few nodes by default, but can be configured for replication.	Fixed schema design where one column contains specific data type and structure is strictly followed.	Supports primary key, foreign key, unique key, not null key and check constraints.	Strong ACID compliance when using InnoDB storage engine.
ArangoDB	Data is distributed across nodes using sharding.	Flexible schema design, supports both document and graph data models.	Supports primary key constraints, no foreign key constraints.	Eventual consistency, can be configured for strong consistency in some cases.
Cassandra	Data is distributed across all nodes and divided into partition using partition key.	Flexible schema design allows each row to have different columns.	Supports primary key constraints, no foreign key constraints and join operations.	Eventual consistency, all the replicas will converge to same value.
Memcached	Data is distributed across multiple servers using a hashing algorithm.	Memcached is a key-value store and does not enforce a schema.	Does not support database-level constraints.	Eventual consistency, focusing on high availability and performance.
MongoDB	The shard key is used to distribute a collection's documents across shards.	Flexible schema model, documents in a collection do not need to have the same fields or data types by default.	Schema validation supports constraints on documents structure.	Eventual consistency and replica sets are used for redundancy and data availability.

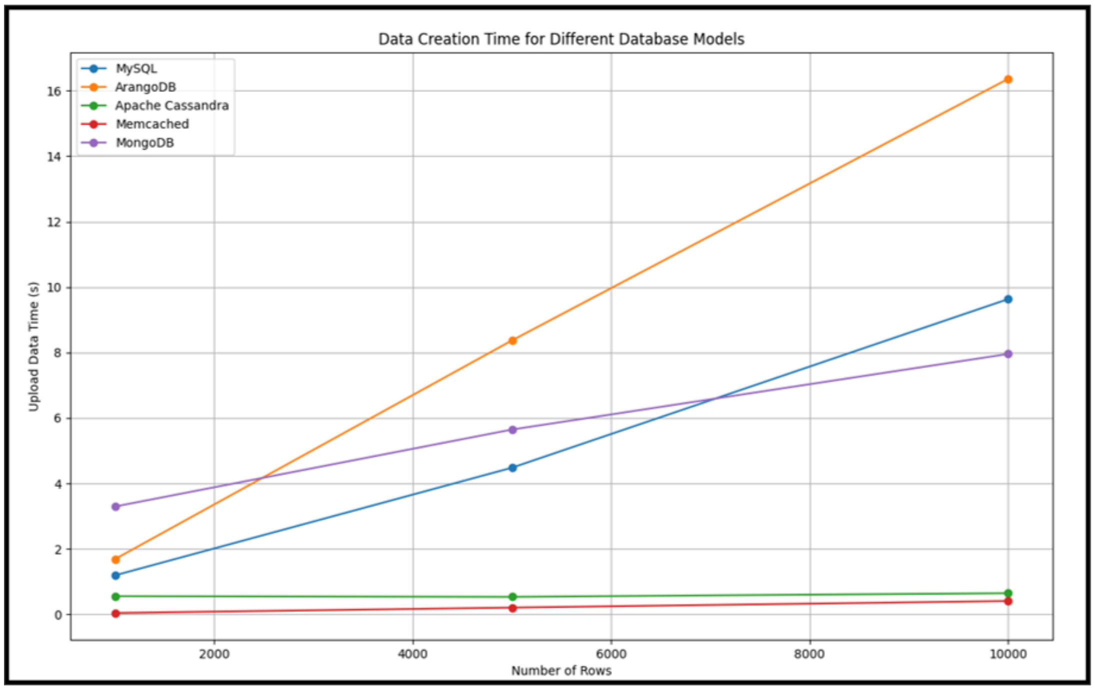


Figure 1: Data Creation Time for Different Database Models.

contexts. This has led to the development of alternative models like NoSQL that provide different approaches to data management and scalability Kunda, D., & Phiri, H., (2017).

Access Control

In MongoDB, access control ensures that only authorized users are able to access and alter data by configuring authentication and authorization MongoDB. (n.d.). Creating authentication and roles, validating setups, and implementing extra data security measures all contribute to effective access control in MongoDB. To enable authentication, start MongoDB with 'auth'. Create an admin user before enabling it:

```
db.createUser({
        user : "admin",
        pwd: "adpwd",
roles:[{ role: "userAdminAnyDatabase",
db: "admin}]
Create custom role:
db.createUser({
        role: "customRole",
        privileges:[{resource:{db: "AnyDatabase",
collection: ""}, actions: ["read"]]},
roles:[] })
```

Data Integrity

Nevertheless, MongoDB is flexible and schema-free, however, it has some drawbacks compared to traditional RDBMS like the absence of data integrity related features shown in Table 3. Different from most of RDBMS that works on the principle of schemas and utilizing primary and/or foreign keys to determine consistency, MongoDB makes use of unique_id fields and implementation of application logics for referential integrity It includes features like, optional schema validation and it seems to support multi-document transactions for atomicity, however, it lacks the full-fledged constraints and strong consistency as offered by relational database management systems. Additionally, MongoDB's eventual consistency model is quite different from the strong consistency offered by centralized RDBMS systems Giamas., (2019).

Summary Table for Each DBMS

Research Methodology

Gathering Information

Our literature review focused on exploring the five primary database models. To support our analysis, we selected specific databases that exemplify each model:

Relational model: MySQL.

Graph model: ArangoDB.

Wide-column model: Apache Cassandra.

Key-value model: Memcached.

Document model: MongoDB.

We aimed to understand how each model operates, particularly in relation to key database operations such as:

Data creation.

Data manipulation.

Data retrieval.

Access control.

Data integrity.

We conducted our literature search using reputable academic platforms, including:

Google Scholar.

Sunway Library.

IEEE Xplore.

ScienceDirect.

Additionally, we consulted official websites and documentation for the selected databases to ensure that our review was informed by accurate and up-to-date information.

Procedures

First of all, we provided an initial survey of each database model with the help of free online courses and YouTube videos to get acquainted with different types of databases and their application.

We then went to the documentation of each of the database models under consideration as an official documentation. For instance, Apache Cassandra has a website that provides descriptions of how it functions and how to use it. Similarly, to understand the specifics of these systems, we referred to the official documentation of MySQL, ArangoDB, Memcached, and MongoDB.

After we had a theoretical understanding of the concepts, we went to the practical implementation of the concepts. For setting up the different conditions and to test the feasibility of the approach, we used the Web resources and trial versions of the databases. For the relational databases like MySQL, we downloaded the MySQL Community Edition and installed it on the local computers for the purpose of testing. For more complex databases such as Apache Cassandra, we utilized virtual machine tools such as Docker to run the database in a container.

To compare the feasibility of each of the database models for the given scenario, we developed a travel booking system. We created sample tables and did simple operations like insertion, update, deletion, select, security and constraints. This practical aspect allowed us to observe how each of the database models handles data and operations in practice.

METHODOLOGY

The research method used in this study was theoretical and practical in an effort to establish the efficiency of the various database models. We began with the fundamental concepts and functions of each database model regarding data input, data modification, data retrieval, security, and data consistency. This theoretical background was useful for the evaluation of the strengths and the weaknesses of each model.

Following this, we conducted hands-on experiments to assess the performance of the selected databases: MySQL, ArangoDB, Apache Cassandra, Memcached, and MongoDB. In these experiments, we generated synthetic data and performed a series of operations that are related to a travel booking system. These operations included insertion of data, update, delete, query, control of access and application of constraints.

To compare the performance and scalability of each of the database models, the time taken to execute each of the queries was measured for different data sizes. We also assessed how each

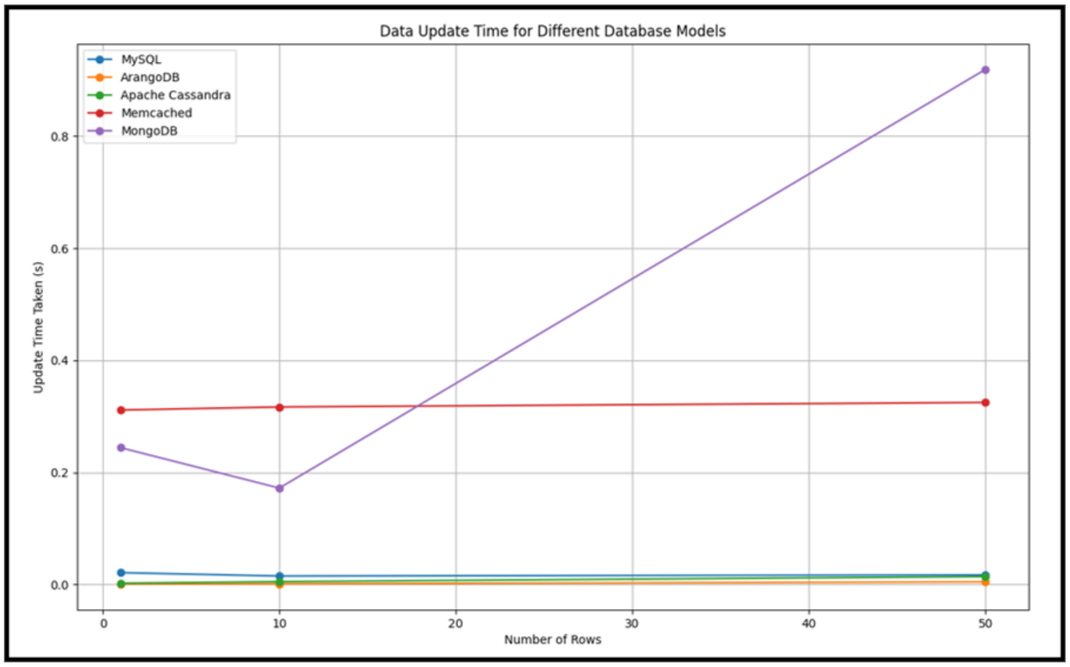


Figure 2: Data Update Time for Different Database Models.

model offered access control and employed constraints to secure data and ensure its accuracy. The collected data was then plotted in order to compare the performance of each model under different conditions. This approach helped to understand how each of the database models works in terms of data volumes, operations, and security, which is useful information on the practical application of the models.

RESULTS

Scenario 1: Flight Booking System

This flight booking system is a complex system which means giving the user as much information as possible about flight schedules and making the process of flight booking easy. An entry in the Flight table involves attributes such as Flight ID, the airline name, departure and arrival cities, the scheduled time of departure and arrival, flight duration, and ticket price. People who wish to travel have the option to search for a particular flight according to their preferred location for departure, destination, airline company, and date on which they intend to travel. Ranges of prices for the flights are available and the flight duration and the time of the flights, whether they are taking off or arriving can also be searched on the system.

Strength and Weaknesses Győrödi, C, *et al.*, (2021)

Relational Database Model (MYSQL)

Strengths

Data integrity

MySQL enforces data integrity through constraints like primary keys, foreign keys, unique constraints and check constraints to ensure data accuracy and consistency.

ACID compliance

MySQL supports ACID (Atomicity, Consistency, Isolation, Durability) properties that guarantee reliable transactions and maintain database stability even in the event of a system failure.

Scalability

MySQL can handle a large amount of data and high traffic that makes it suitable for both small-scale and enterprise-level applications with the ability to scale horizontally.

Query optimization

MySQL includes a query optimizer that determines the most efficient way to execute SQL queries, improving the performance and speed of data retrieval.

Security

MySQL incorporates strong security features like user authentication, SSL encryption as well as access control to ensure that data is safeguarded against unauthorized access and breaches.

Weaknesses

Complexity

Achieving optimal performance and security may need complex configuration and fine-tuning that can be challenging and time-consuming for inexperienced users.

Cost

Even though MySQL itself is open-source, certain advanced features and support services may come at a cost, particularly with enterprise versions like MySQL Enterprise Edition.

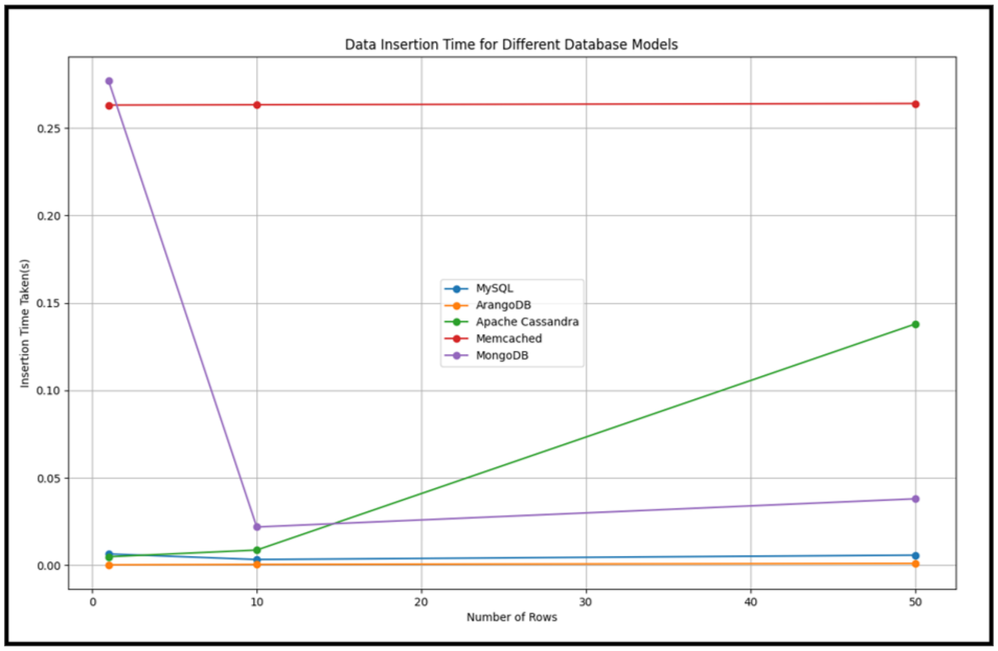


Figure 3: Data Insertion Time for Different Database Models.

Scalability

While MySQL can handle significant loads, it may face some scalability issues in extremely high-transaction conditions that require careful configuration and optimization.

Performance

MySQL can experience performance bottlenecks, especially with complex queries, large join operations, or high-transaction volumes, which may require extensive optimization.

Limited NoSQL features

MySQL's support for NoSQL features such as JSON data types and document storage is not as advanced as dedicated NoSQL databases. This limits its flexibility for certain types of data and applications.

Graph Model (ArangoDB)

Strengths

Multi-Model Capabilities

This makes it easy to work with as it supports documents KB and graphs and ACID transactions. This is especially useful in a flight booking system since the different entities such as flights, passengers, bookings, and so on, can well represented.

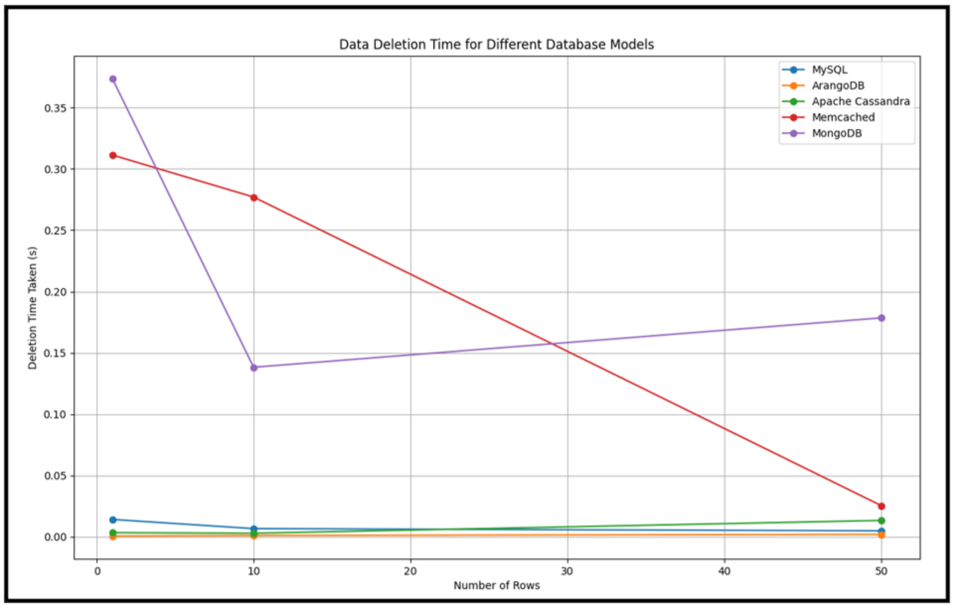


Figure 4: Data Deletion Time for Different Database Models.

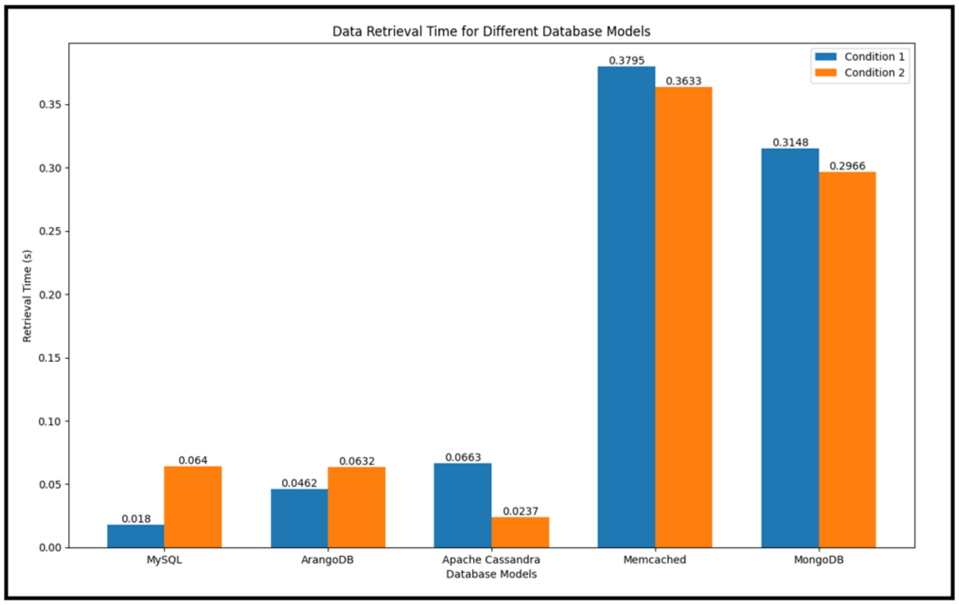


Figure 5: Data Retrieval Time for Different Database Models.

AQL (ArangoDB Query Language)

AQL offers similar querying functionality that allows for more complex queries such as flight search, book flights, or access to the passenger details. Learning it is easier for developers who have worked with relational databases; this is due to its resemblance to SQL.

Scalability

ArangoDB is horizontally scalable and will be able to handle large amounts of data and traffic characteristic of a flight booking system. Sharding/ partitioning and replication is another feature which aims at making a provision for expansion of the database with growth of the demands of the systems.

Performance

One major factor is the database's capacity to handle such elaborate queries because response time is a big aspect in a flight booking system. ArangoDB's operation in terms of querying and indexing is a plus for keeping the system on the right side of response time.

Flexibility

This structure of ArangoDB is favourable for models change and extension for it does not require changes of the whole structure of the database when business requires.

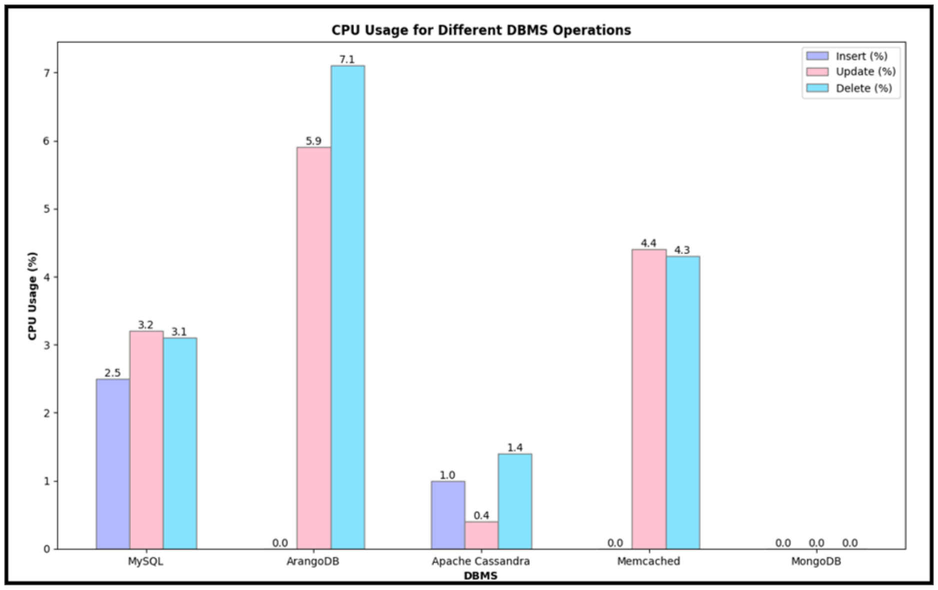


Figure 6: CPU Usage for Different DBMS operations.

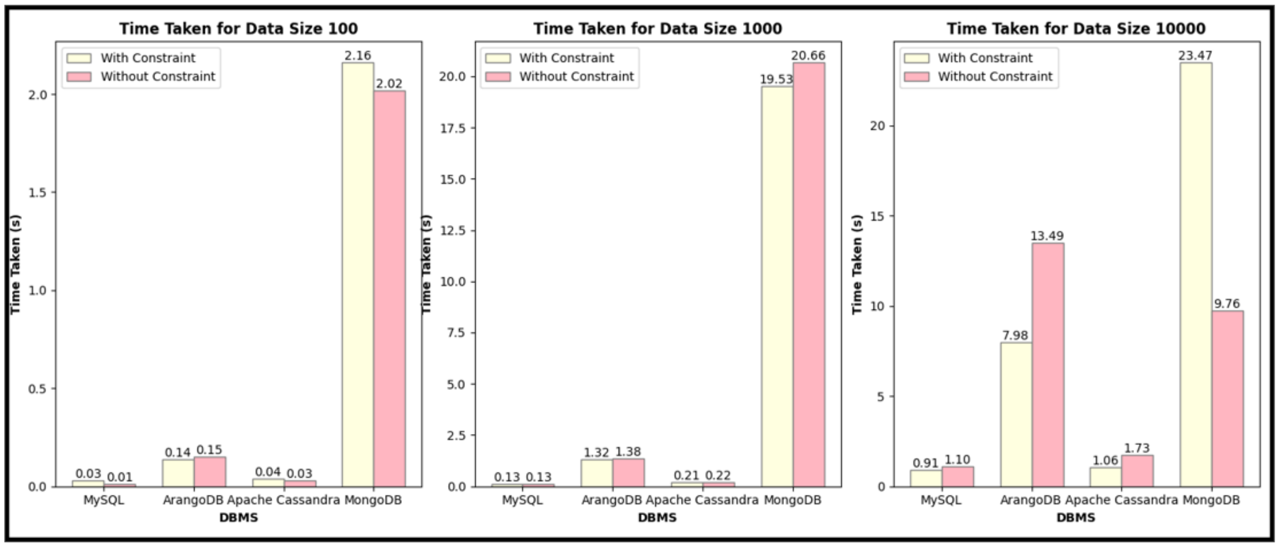


Figure 7: Different DBMS Operations *with* and Without Constraints on Different Data Size.

Weaknesses

Learning Curve

In common, AQL is very useful, but in the same time, it needs a learning curve for those who did not work with ArangoDB system and its concept of the multi-model database. This might mean that the development team needs more training in the use of the defined process.

Complexity of Multi-Model Management

Multiple data models might complicate the structure and designs of the information systems. The work that developers do requires them to be familiar with different types of data and how those data can be integrated.

Community and Ecosystem

Thus, ArangoDB is less popular than databases like MongoDB or MySQL: the number of users and the support for this system

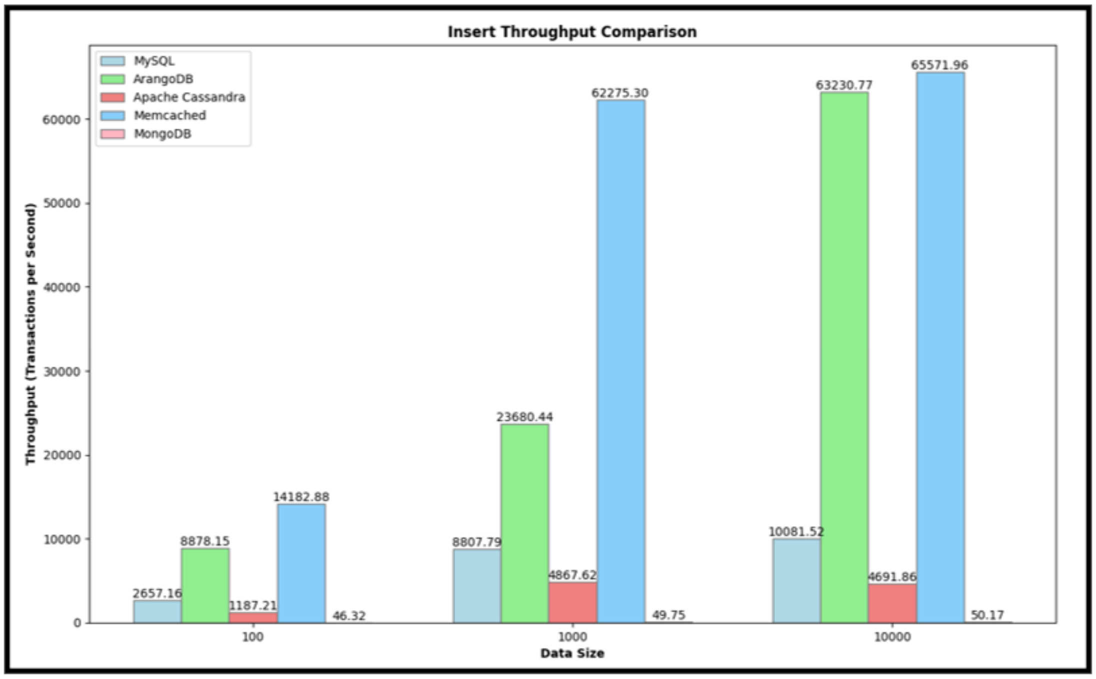


Figure 8: Insert Throughput Comparison Between Different Database Models.

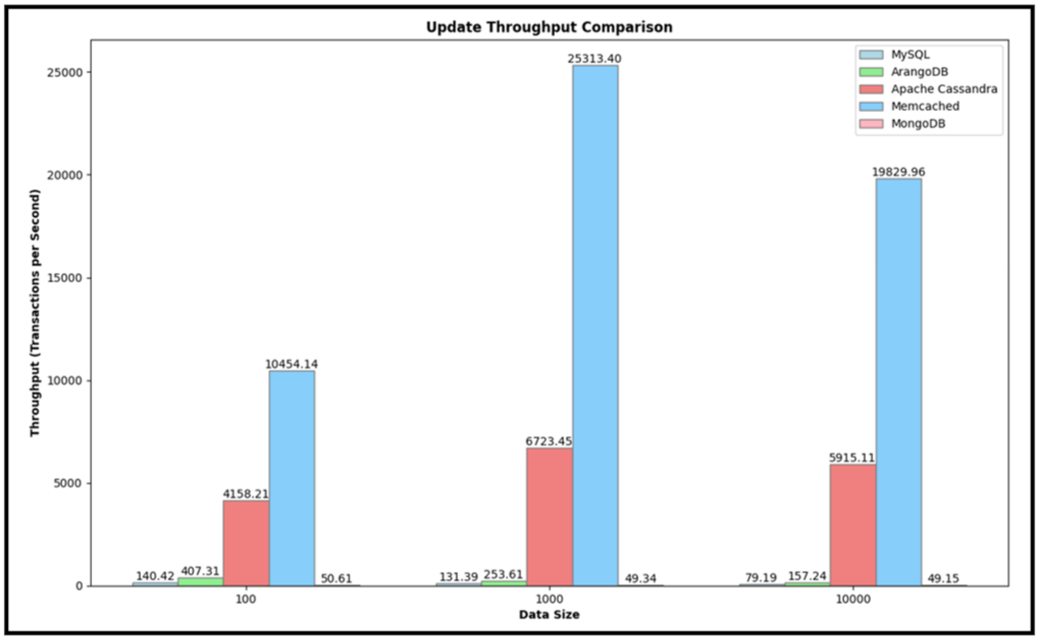


Figure 9: Update Throughput Comparison Between Different Database Models.

is much lower. This can lead to fewer options for getting support and integrating different kinds of tools, resources and plugins

Enterprise Features

Certain additional features including security options, the ability to scale the size of the site and additional administrative functions are restricted to the paid for Enterprise option which may prove more costly for businesses who require such additions.

Tooling and Integration

Even though ArangoDB has its hookups with numerous tools and platforms, it emerges that the support may not be as profound as one would expect given other databases. There is always a possibility that the chosen solution will have to be adapted to fit the existing infrastructure as well as interfaced with third-party tools and services, which would take extra work.

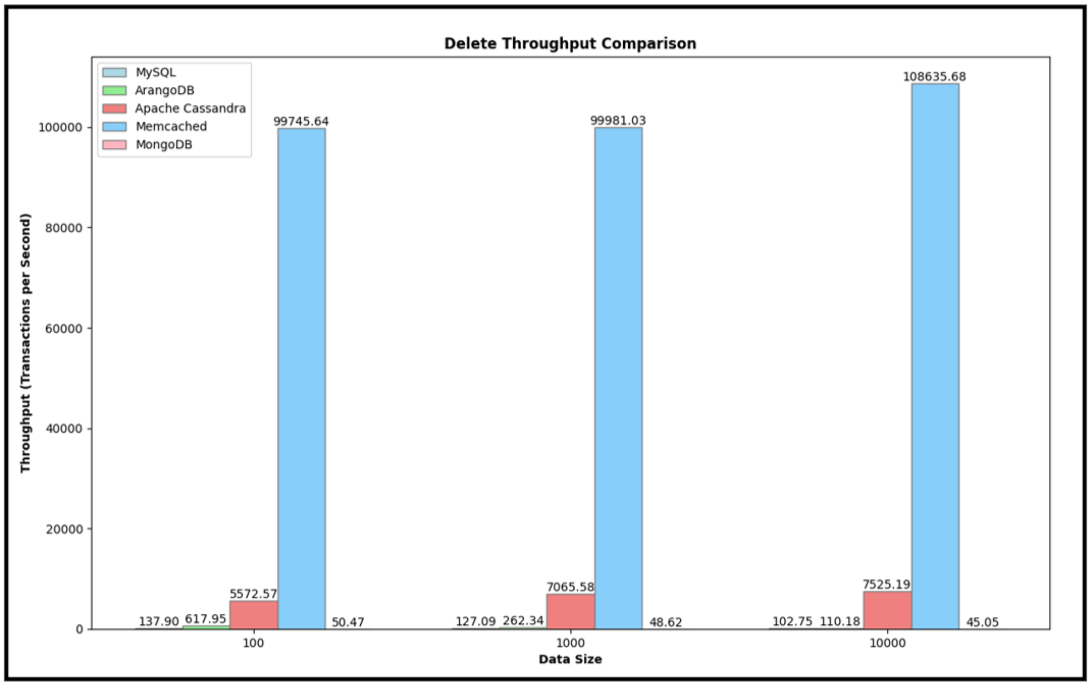


Figure 10: Delete Throughput Comparison Between Different Database Models.

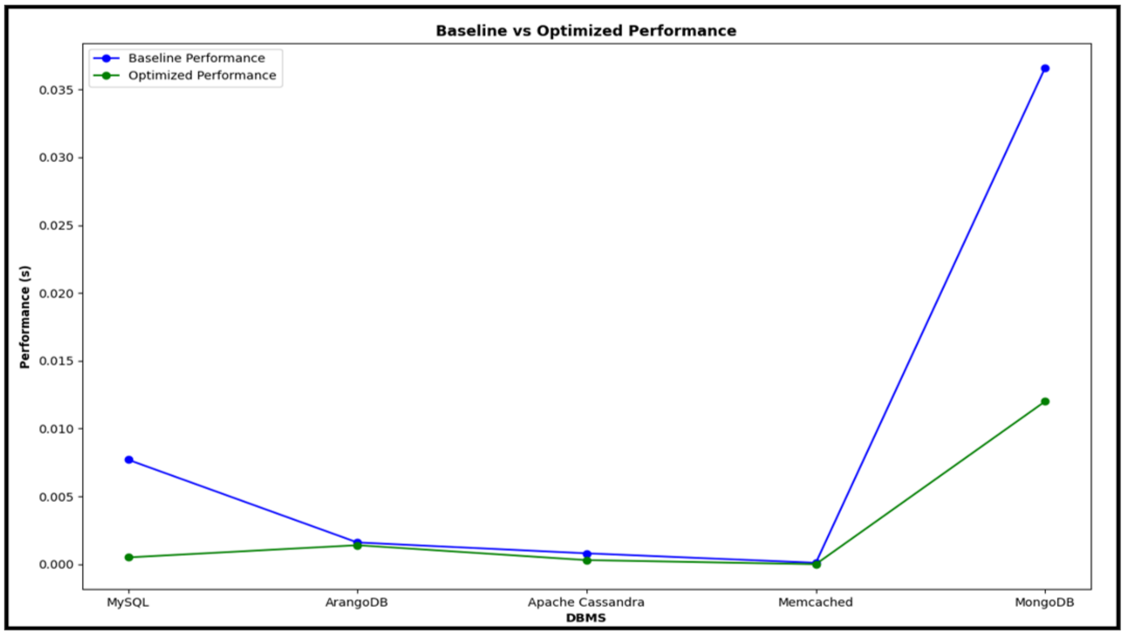


Figure 11: Difference between Baseline Performance vs Optimized Performance.

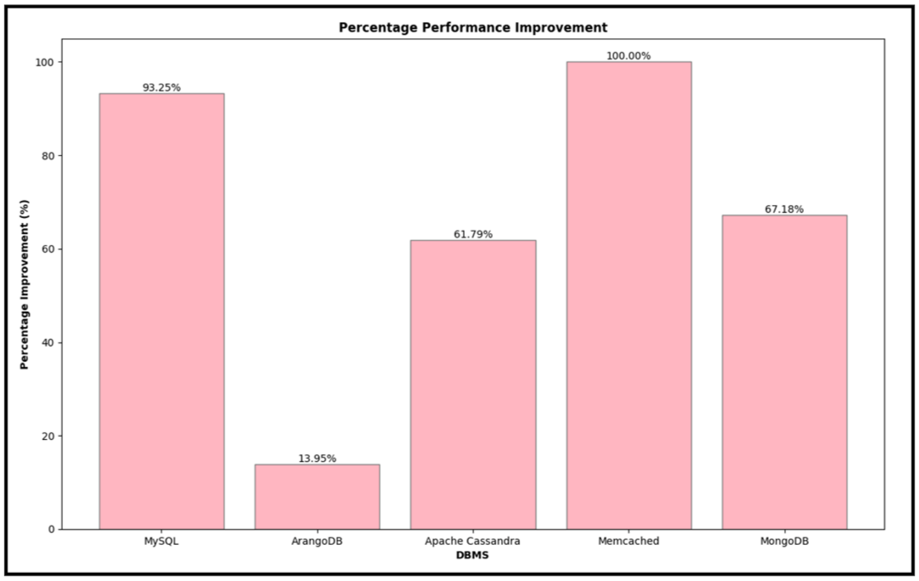


Figure 12: Performance Improvement of Different Database Models in Percentage.

Table 4: Two Condition for Data Retrieval.

Condition 1:	Condition 2:
Retrieve flights where the price is between \$200 and \$500, and the destination is Dallas.	Retrieve flights operated by "Airline A" or "Airline B" that depart from "New York" and arrive in "Los Angeles".

Wide column model (Apache Cassandra)

Strengths

Scalability

In Cassandra, it enables horizontal scaling to be performed very easily. This helps the system to respond to high volumes of data creation requests like flight bookings and customers registration while using the number of nodes and no node is at a failure point.

High write throughput

Cassandra is designed for high write throughput as it is useful for systems where data is frequently written such as changing booking status or flight schedules. This is due to the fact that its log-structured storage is append-only, which makes the writing operation fast and efficient.

Low latency reads with proper data modeling

When data is modeled correctly, Cassandra is capable of delivering low latency read by being able to find the data in the right nodes using the partition key. This is helpful when searching for flight schedules, booking information and customer data.

Role-Based Access Control (RBAC)

Cassandra supports role-based access where privileges are granted based on roles, and these roles are defined with certain access rights. This assists in the protection of the data and equally allows only the user who is permitted to do so.

Lightweight Transactions (LWT)

LWT in Cassandra is a way to enforce strong consistency for certain operations, such as conditional update or insert. This is especially helpful in avoiding such cases happening when two people being assigned to the same seat.

Weaknesses

Lack of complex constraints

Cassandra is limited in its ability to support complex constraints such as foreign keys, unique constraints other than primary keys, and check constraints. This implies that some of the data integrity constraints, for example checking that each customer has a unique email address for checking that a seat has not been double booked has to be checked at the application level rather than at the database level.

Eventual consistency

It is important to note that all the data manipulation operations in Cassandra are eventually consistent by default. This means that there may occur a time lag before all the nodes in the cluster have updated data and when this happens, the data will be inconsistent temporarily. This is especially the case when there needs to be a very strict level of consistency, such as during payment or booking seats.

Limited query flexibility

Cassandra expects the queries to be built around the partition key or the secondary indexes. Without these, data retrieval becomes either inefficient or is not supported at all. The 'ALLOW FILTERING' clause can be used but should be noted that this can cause performance variability and high resource usage.

Limited granular control

RBAC is useful for securing the database, but Cassandra's access control mechanisms may not be as refined as the ones in traditional rational database systems. For instance, features like row-level security are not implemented, which may be a problem when working with multi-tenant applications or any other cases when data access must be carefully controlled.

Limited support for complex integrity constraints

Cassandra does not support some of the advanced integrity constraints such as foreign key and unique constraints which may make it somewhat difficult to enforce certain data integrity rules at the database level. This often requires more application processing or invokes other processes to validate the data and ensure data integrity.

Key-Value Model (Memcached)

Strengths

High Performance

Memcached is designed for speed, providing quick data retrieval and storage, making it ideal for caching frequently accessed data to improve application performance.

Scalability

It can be scaled horizontally by adding more nodes to the cluster, allowing it to handle increased loads and large datasets effectively.

Simplicity

Memcached has a straightforward design and easy-to-use APIs, making it simple to integrate and use within applications.

Memory Efficiency

It has a good memory management since it has a mechanism of Removing the Least Used (LRU) to allow efficient use of the available memory.

Wide Adoption and Community Support

Memcached is quite popular and has a lot of backing, which means that there is a lot of information available on the internet and many people who can help if you run into problems.

Weaknesses

Lack of Persistence

Memcached does not support persistence; all the data stored in it is lost in case of server crash or restart, which can be a major disadvantage for some applications.

Limited Data Types

It mainly works with simple key-value data and does not support other types of data and operations as in Redis.

No Built-in Security Features

Memcached does not have built-in authentication and encryption, so data protection must be implemented separately, for example, by connecting to a VPN or using it in a secure network.

Eventual Consistency

Memcached follows an eventually consistent model, which means data consistency across nodes is not guaranteed immediately, potentially leading to stale data being read under certain conditions.

Memory Overhead

Memcached can have significant memory overhead due to metadata storage and slab allocation, which can reduce the available memory for actual data storage, especially with small items.

Document Model (Mongodb)

Strengths

Flexible schema

The flexibility of schema in MongoDB allows adding fields or changing the structure of documents without predefining a schema which is useful for applications where data requirements expand over time.

Atomic operations

At document level, atomic operations in MongoDB is allowed to ensure complete execution of changes to single documents.

Aggregation framework

The aggregation framework in MongoDB supports processing of complex data and transformation in database, and the need for substantial client-side processing is minimized.

Granular permissions

Users' permissions can be set at any level which includes databases, collections and the operations of users. This gives pulverized control on data access.

Replication and sharding

Enhancement of data integrity and availability in MongoDB is by its built-in replication which is for data redundancy and the sharding that is for horizontal scaling mechanisms.

Weaknesses

Weak data validation

In MongoDB, the schema validation is different from other RDBMS as it is not as rigid which means that consistency of data during creation might be less.

Limited multi-document transactions

Although MongoDB has implemented multi-document transactions 4.0, but it is not as advanced or efficient as those offered in traditional RDBMS. This may have an impact on manipulation of complex data operations including several documents.

Performance with large datasets

When MongoDB is dealing with large datasets, the performance of its querying and retrieval might be downgraded if indexing is not properly done. With proper use of indexing, MongoDB will perform well that means MongoDB is dependent on indexing.

Default security settings

Data in MongoDB might be left vulnerable if the default security settings are not changed accordingly. Therefore, it is important to obey the best practices while securing MongoDB deployments.

Lack of ACID compliance

MongoDB does not provide full ACID compliance over multiple documents and collections even though it supports several levels of transaction. This means that data consistency might be affected in some cases.

Comparison of each Models in Scenario 1

Data Creation

Based on Figure 1, Apache Cassandra and Memcached indicate the least time taken to import data when the number of rows increases. This makes them suitable for situations where the data insertion rate is important. On the other hand, MySQL and MongoDB demonstrate that the time taken to create data is moderately higher than in the other databases. While ArangoDB takes the longest time and thus is not very useful in situations where data generation is needed in a short time.

In addition, the most suitable database model for a flight booking system is Apache Cassandra since it has high scalability and only with minimal impact on creation time. Memcached can also be incorporated as a caching layer to increase performance. MongoDB is also a consideration if flexibility in the schema is a concern. MySQL and ArangoDB are less preferable as data creation time is relatively slower in case of high load.

Data Manipulation

Update Data Operation

According to Figure 2, when it comes to DBMS for a flight booking system that needs frequent data updates, some of the best options are Apache Cassandra, MySQL and ArangoDB that offer minimal and constant update time that is perfect for high performance and scalability. Memcached also behaves reasonably well but it is recommended to be used as a caching layer. However, for MongoDB, the update times are increasing when rows increasing, it is not as efficient for scenarios that require frequent and fast updates. In general, the best performance in managing real-time data updates is offered by Apache Cassandra, MySQL and ArangoDB.

Insert Data Operation

Based on Figure 3, MySQL and ArangoDB have the lowest insertion times across all the tested row size. The insertion time of Memcached is the highest and constant depending on the number of entries, thus making it unfit for frequent insertion. Apache Cassandra and MongoDB exhibit different results; for Cassandra, the insertion time increases with the number of rows, while for the MongoDB, the insertion time is high at the beginning but level off afterward. In a scenario of flight booking system, where there is a high frequency of insert operations, MySQL and ArangoDB are the most appropriate since they have the least insertion time. MySQL may be chosen if the system uses its strong relational database capabilities, ArangoDB may be suitable if a more free-form and multi-model DBMS is required.

Delete Data Operation

Figure 4 shows that for a flight booking system that needs to perform efficient and accurate deletions, Apache Cassandra, MySQL and ArangoDB are the most suitable because they have low and fairly stable deletion time even the number of rows increases. The trend of deletion time in Memcached is decreasing, which means that though deletion time may have an initial overhead, it can be managed and used for specific caching requirements. While MongoDB is not as consistent as Redis in terms of deletion time, as it has higher initial deletion time, and is not very constant. Therefore, for reliable and efficient deletion of data, Cassandra, MySQL and ArangoDB are the most recommended databases.

Data Retrieval

According to Figure 5, Regarding condition 1, fast retrieval times are essential for filtering based on the price range and destination, in which MySQL is in the first place based on the minimum retrieval time with ArangoDB and Apache Cassandra close behind. On the other hand, condition 2 involves filtering by multiple attributes include airline, departure city and arrival city. Apache Cassandra is the best option in terms of the lowest retrieval time of 0.0237 sec, it is very effective especially when it comes to queries that involve many attributes. Thus, although MySQL is mostly effective, Cassandra is more effective in specific multi-attribute retrieval cases, thereby stressing on the necessity of choosing a database model depending on the query type. While MongoDB retrieval times are higher than MySQL, ArangoDB and Cassandra but slightly lower than Memcached, however Memcached has the highest retrieval time among all the database for both different conditions. Table 4 Shows two different conditions for the data retrieval purpose.

CPU Usage

Based on Figure 6, distinct performance characteristics of each database models will be revealed through insert, update and delete operations. MySQL shows moderate CPU utilization over all operations, including insert operations at 2.50%, update operations at 3.20%, and delete operations at 3.10%. This implies that MySQL has a balanced strategy to allocate resources. ArangoDB executes no CPU utilization for insert operations, but more percentages for update operations at 5.90% and delete operations at 7.10%. This indicates that ArangoDB requires intensive processing on these operations. Apache Cassandra shows low CPU usage, including 1.00% for insert operations, 0.40% for update operations and 1.40% for delete operations, indicating Apache Cassandra's efficiency on handling these operations. In the context of Memcached, the CPU utilization is relatively low for zero usage during insert operations, however, higher for update operations at 4.40% and delete operations at 4.30%, which show that Memcached is relied on specific caching mechanisms which its performance over these operations will be affected. With zero CPU utilization over these operations, MongoDB stands out with its high efficiency of processing. To sum up, Figure 6 showed the efficiency of each model during insert, update and delete operations.

Data Integrity

Among the databases in Figure 7, MySQL consistently performs well displaying minimal differences between operations with and without constraints. As an example, at 100 data size, MySQL takes 0.03 seconds with constraints and 0.01 seconds without, hence indicating high efficiency. On the contrary, MongoDB exhibits the least performance with constraints. In data size of 10,000, MongoDB takes 23.47 seconds with constraints compared to 9.76 seconds without constraints that underline a substantial overhead when constraints are applied. ArangoDB also shows an increase in time with constraints at higher data sizes, with 7.98 seconds with constraints and 13.49 seconds without constraints for 10,000 data sizes. Apache Cassandra shows stable performance with minimal differences between constrained and unconstrained scenarios which then proves its efficiency to handle constraints efficiently. In short, MySQL and Apache Cassandra perform well under constraints whereas MongoDB and ArangoDB face significant slowdowns as the data size increases.

Scalability

Insert Throughput Comparison

According to Figure 8, ArangoDB's and Memcached's insert throughput excels among the 5 database models. This is due to their consistent increase in insert throughputs across data size grows. This is critical as flight booking system requires a lot of insert operations. For MyS QL and Apache Cassandra, their insert throughput also increases with data size growth, however, MySQL and Apache Cassandra could not handle as much insert operations as ArangoDB and Memcached. For MongoDB, due to its limited capacity for handling insert operations, its insert throughputs are significantly low but consistent.

Update Throughput Comparison

Based on Figure 9, although Memcached leads among the other database model, but there is a significant decrease as data size grows. In this case, Apache Cassandra shows its consistency in update throughputs as data size grows. For MySQL, ArangoDB and MongoDB, their update throughputs decrease across the growth of data size. Overall, Memcached could possibly be the optimal model for update throughputs as there is many update operations in a flight booking system.

Delete Throughput Comparison

Figure 10 shows the differences between the delete throughput performance of the database models. Memcached excels among the other database models as it peaks at 108635.68 operations per second for 10000 data size. Upcoming is the Apache Cassandra with consistent efficiency even data size is growing. For MySQL, ArangoDB and MongoDB, they showed a significantly low efficiency of delete throughputs as compared to Memcached and Apache Cassandra. Overall, Memcached excels in delete operations which plays a vital role in a database.

Performance Improvement

Figure 11 shows the difference between baseline performance and optimized performance for each database models. Based on Figure 12, the enhancements in different databases investigated due to various focused enhancements employed for improving the specific system. MySQL has achieved a 93.25% performance improvement possibly from better indexing, query processing and table access. ArangoDB demonstrated a 13.95% performance improvement indicating that it had been uniquely optimized beforehand. Apache Cassandra's 61.79% performance improvement is likely to arose from the improvement of data dissemination and indexing techniques. In the case of Memcached, there is a magnificent improvement of 100.00% as its real caching in memory, thus eliminating time for data retrieval. MongoDB achieved a 67.18% performance improvement which is likely to be attributed to improved indexing techniques, query restricting, and comprehensiveness in dealing with huge quantities of data. Such optimizations include improving the indexing process, rewriting the queries, efficient portioning of data, using cache mechanisms, improving the physical hardware, and optimal configuration of hardware which results in efficient access to data and improved performance of the database.

DISCUSSION

Data Creation

The analysis of the results reveals that there are major differences in the performance of data creation between different database models. Thus, the most effective and highly scalable systems are Memcached and Apache Cassandra as both are used for handling huge amounts of data which is useful in the flight booking system scenario where data transactions are frequent and massive. The higher efficiency of these NoSQL databases indicating that the traditional relational database such as MySQL may not be as ideal for high throughput applications, although they are very reliable with well-defined structures.

This study supports existing literature on the applicability and effectiveness of the NoSQL databases and especially for the scenarios that require high performance and fast data processing. Prior research has shown the benefits of NoSQL databases in the context of big data technologies Janjua, J. I., *et al.*, (2022), and the results of this paper support the practical utilization of NoSQL databases in real-time transactional systems such as flight booking system. The findings of this research enrich the existing literature on the context-specific benefits of various databases.

Data Manipulation

From the result of the study, Cassandra, MySQL and ArangoDB stand out as better suited for handling real-time updates and demonstrate the best performance in delete operations. MySQL and ArangoDB are especially suitable because of their relatively low insertion time. Hence, these results indicate that the selection of the right database can greatly affect the speed and accuracy of data manipulation operation, which is essential in real-time applications such as flight booking systems.

The contribution of this research work to the current knowledge base is the empirical feature that measures the performance of the various database in data manipulation activities. Prior studies have pointed out that NoSQL databases are well suited for massive transactions Leavitt, N., (2010). We can confirm these claims with our result especially in the case of MySQL and ArangoDB excel in insert, update and delete operations. This study also show that MySQL is still useful in cases where the relational strength is demanded.

Integrity

As noted in McMinn, P., *et al.*, (2015), the impact of constraints on performance varies significantly across different systems that underlines the importance of tailored database strategies. The study's findings reveal that MySQL outperforms other database management systems in both constrained and unconstrained scenarios that proves its superior efficiency to maintain data integrity while delivering consistent performance.

We have learned that MySQL is highly suitable for applications requiring extensive data accuracy without compromising speed. Conversely, MongoDB showed the longest processing times, particularly under constraints that indicate substantial challenges to balance data integrity with performance. This performance degradation suggests that while MongoDB offers flexibility and scalability, it may not be the ideal choice for applications with strict data integrity needs. Our study contributes to existing knowledge by underlining these performance differences and reinforcing the importance of selecting a DBMS based on specific requirements.

Query Performance

As applied to the enhancement of database efficiency, this study has several implications. The degree of performance boost for the three database models like MySQL, Apache Cassandra and MongoDB, despite the fact that the degree if tuning as seen from the query optimization has not been very high in most cases, targeted optimizations and data distribution have proven to work very well. From the results, it is evident that with proper optimization, even matured databases have the potential of getting a big performance boost, thus enhancing their capability in terms of real-time processing of ever-growing datasets. The results also show the effectiveness of in-memory databases such as Memcached for some cases when data access speed is critical but at the same time, there is a limitation on the complexity of the queries to be performed.

This study is based on the literature review, carrying on research by presenting the experiment-based data on the effects of optimisation algorithms on a number of relational and non-relational DBMS. Previous works have been concentrated in isolated parameters of database performance. However, those aspects are holistically synthesized in this study as an optimization approach and demonstrate how synergistic initiatives yield enhances performance. The study adds to the research finding done in the past like positive impacts of efficient indexing or proper allocation of index table Nambiar, R., & Poess, M., (2011).

Scalability

This research sheds some light on an important issue which is the case of choosing the right DBMS depending on given performance criteria especially in growth-oriented organizations. The exceptional performance for insert, update and delete of Memcached shows that it is very efficient for high throughputs of data in the applications like the caching system, real-time data processing applications and large-scale web applications. These results corroborate the need to determine the peculiarities of various DBMS choices when it comes to developing systems with high scalability demands.

Thus, this research completed the existing knowledge stock as it provides a comparative study of how the DBMS under different sized of the data works and operates. Previous work can be characterized as mainly concentrating the individual parameters or certain kinds of the databases. This work reveals the and enlarges the knowledge on how DBMS like MySQL, ArangoDB, Cassandra, Memcached, and MongoDB perform multiplexer operation by experimenting numerous operations such as insert, update and delete. The outcomes are consistent with other studies that separate Memcached's effectiveness during high-speed operations stemming from its in-memory caching framework and its capacity to scale when facing with large amounts of data Bermbach, D., Wittern, E., & Tai, S., (2018).

CONCLUSION

In conclusion, based on the analysis, Apache Cassandra emerges as the most appropriate database model for flight booking system scenario. It also performs well in terms of the high throughput operations, which are important in a real-time booking system. Due to its ability to scale horizontally, have high availability, and able to handle large amounts of transactions efficiently, Cassandra is well suited for flight booking system that are dynamic and complex. Besides that, MySQL and ArangoDB have distinctive strong performance in certain aspects; however, Cassandra has a more stable and comprehensive performance in most of the aspects in handling a large amount of data.

Furthermore, this project has made our team understand the need to choose the right database that would help in making the operation of data in a system more efficient. By comparing performance in different data sizes, it was clear that databases such as Cassandra and Memcached are very efficient when it comes to handling large data sets and performance consistency. This means that the selection of DBMS should depend on the application needs and the anticipated size of data rather than going for traditional databases.

Our results also highlighted the significance of optimizing the functionality of the DBMS to match the application requirements for enhancing the performance. The study showed that choosing the right database is crucial for determining the nature and capabilities of an application. MySQL and ArangoDB were excellent in certain operations, whereas, Memcached was excellent in handling huge amounts of data. These insights underscore the fact that the DBMS decisions should be made depending on the operational characteristics and data processing needs of the application for achieving the best performance and scalability.

Besides that, there are several limitations found in the study. The performance tests environment was controlled and this implies that some of the values may differ from a real world environment where some elements such as network delay, differences in the user interactions and the workloads may differ. The study mainly concentrated on the retrieval time while other essential measures like write time and systems capability under different loads were not considered. In addition, the hardware and software environments that might have been used may not be similar to those that are used in real life by these databases, which reduces the external validity of the studies. The limitations of the study were also present in the focus on a particular testing scenarios, which may not represent all possible cases.

As for the limitations of the current research, future research should consider the areas that were beyond the scope of this research. First, studies of databases under real-world conditions will give a more realistic view of the databases' behaviour, taking into account factors such as network delay and varying load. Furthermore, it is important to investigate the efficiency of optimization techniques that are applied to specific kinds of databases because each DBMS has its strengths and weaknesses as well as specific critical points. In addition, implementing machine learning algorithms that can predict data quality and its optimal conditions in real-time can improve the quality and reliability of databases under different circumstances. These areas will be addressed to eliminate the current study's drawbacks and gain more profound insights into the best practices in database optimization.

Group member contributions

Student Name (ID)	Percentage of contribution in each section (1-5)	Percentage of the overall participation
Chan Pui Ying 21041546	1 (35%), 2 (20%), 3 (35%), 4 (5%), 5 (5%)	100%
Joselyn Chin Shi Min 22049126	1 (10%), 2 (20%), 3 (10%), 4 (30%), 5 (30%)	100%
Gan Yi Jean 22041321	1 (35%), 2 (20%), 3 (35%), 4 (5%), 5 (5%)	100%
Leong Zheng Xuan 21026976	1 (10%), 2 (20%), 3 (10%), 4 (30%), 5 (30%)	100%
Nurjiha Natasha Binti Md Rafi 21054804	1 (10%), 2 (20%), 3 (10%), 4 (30%), 5 (30%)	100%

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest.

ABBREVIATIONS

RDBMS: Relational Database Management System; **SQL:** Structured Query Language; **NoSQL:** Non-Structured Query Language (used informally to describe non-relational databases);

ETL: Extract, Transform, Load; **ACLs:** Access Control Lists; **JSON:** JavaScript Object Notation.

SUMMARY

This paper conducts a comparative analysis of relational and non-relational database models using a case study of a travel booking system. It highlights the evolution from traditional RDBMS like MySQL, which offer structured storage and strong integrity constraints, to NoSQL alternatives such as ArangoDB, Cassandra, Memcached, and MongoDB that provide greater flexibility and scalability for unstructured and big data environments. The study evaluates the two database paradigms across five dimensions: data creation, manipulation, retrieval, access control, and integrity. Findings suggest that while relational databases are suited for consistency and structured data, NoSQL databases offer performance advantages and adaptability in dynamic and large-scale data applications. The insights are particularly valuable for system architects and IT decision-makers aiming to select the optimal database model for modern applications.

REFERENCES

Alflahi, E., Mohammed, M. A. Y., & Alsammani, A. (2023). Enhancement of database access performance by improving data consistency in a non-relational database system (NoSQL). arXiv.org. https://arxiv.org/abs/2308.13921

Bansal, S. K., & Kagemann, S. (2015). Integrating big data: A semantic extract-transform-load framework. https://keep.lib.asu.edu/items/129150. Computer, 48(3), 42–50. https://doi.org/10.1109/MC.2015.76

Bermbach, D., Wittern, E., & Tai, S. (2018). Cloud service benchmarking: Measuring quality of cloud services from a client perspective. Springer.

Blansit, B. D. (2006). The basics of relational databases using MySQL. Journal of Electronic Resources in Medical Libraries, 3(3), 135–148. https://doi.org/10.1300/J383v03n03_10

Callan, J. (2005). Distributed information retrieval. In (pp. 127–150) Kluwer Academic Publishers eBooks. https://doi.org/10.1007/0-306-47019-5_5

Cassandra – Create table. https://www.tutorialspoint.com/cassandra/cassandra_create_table.htm

Cattell, R. (2011). Scalable SQL and NoSQL data stores. ACM SIGMOD Record, 39(4), 12–27. https://doi.org/10.1145/1978915.1978919

Chen, J.-K., & Lee, W.-Z. (2019). An introduction of NoSQL databases based on their categories and application industries. https://www.semanticscholar.org/paper/An-Introduction-of-NoSQL-Databases-Based-on-Their-Chen-Lee/d558abb388590f3db13969a790e002617c8844b7. Algorithms, 12(5). https://doi.org/10.3390/a12050106

Create a user – MongoDB manual v7.0. https://www.mongodb.com/docs/manual/tutorial/create-users/#std-label-create-users

Create KEYSPACE | CQL for cassandra 3.0. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlCreateKeyspace.html

Create role | CQL for cassandra 3.0. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlCreateRole.html

Daugirdas, D., & Zatorskis, J. (2023). Creation of the database prototype. https://www.semanticscholar.org/paper/Creation-of-the-database-prototype-Daugirdas-Zatorskis/25f5caef14ccca97cf8a2dba14bc9f7b1286f874

Documents – MongoDB manual v7.0. https://www.mongodb.com/docs/manual/core/document/#insert

Elmasri, R., & Navathe, S. (2013). Fundamentals of database systems.

GeeksforGeeks. (2022). Key-value data model in NoSQL. https://www.geeksforgeeks.org/key-value-data-model-in-nosql/

Giamas. (2019). Mastering MongoDB 4. x: Expert techniques to run high volume and fault-tolerant database solutions using MongoDB (2nd ed.), 4 p. x. Packt Publishing Ltd.

Győrödi, C. A., Dumșe-Burescu, D. V., Győrödi, R. Ș., Zmaranda, D. R., Bandici, L., & Popescu, D. E. (2021). Performance impact of optimization methods on MySQL document-based and relational databases. Applied Sciences, 11(15), 6794. https://doi.org/10.3390/app11156794

Janjua, J. I., Khan, T. A., Zulfiqar, S., & Usman, M. Q. (2022). An architecture of MySQL storage engines to increase the resource utilization. https://doi.org/10.1109/balkancom55633.2022.9900616

Kunda, D., & Phiri, H. (2017). Comparative study of NoSQL and Relational database. Zambia ICT Journal, 1(1), 1–4. https://doi.org/10.33260/zictjournal.v1i1.8

Leavitt, N. (2010). Will NoSQL databases live up to their promise? Computer, 43(2), 12–14. https://doi.org/10.1109/MC.2010.58

Mcminn, P., Wright, C. J., & Kapfhammer, G. M. (2015). The effectiveness of test coverage criteria for relational database schema integrity constraints. ACM Transactions on Software Engineering and Methodology, 25(1), 1–49. https://doi.org/10.1145/2818639

Mongo, D. B. MongoDB CRUD operations. https://www.mongodb.com/resources/products/fundamentals/crud

MongoDB CRUD operations – MongoDB manual v7.0. https://www.mongodb.com/docs/manual/crud/#delete

Multi model – ArangoDB. (2024). https://arangodb.com/multi-model/

Nambiar, R., & Poess, M. (2011). Performance evaluation and benchmarking: Second TPC Technology Conference, TPCTC 2010. Springer.

Parate, R. (2023). NoSQL Databases: Facebook case study and Analysis. https://www.semanticscholar.org/paper/NoSQL-Databases%3A-Facebook-Case-study-and-Analysis-Parate/4de2d4ae44268dadd1005bdad12270fa880e08f6

Petković, M., & Jonker, W. (2007). Security, privacy, and trust in modern data management. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-69861-6

'Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool.' Seghier, N.B., and Kazar, O. (2021). Select. CQL for Cassandra 3.0." https://www.semanticscholar.org/paper/Performance-Benchmarking-and-Comparison-of-NoSQL-vs-Seghier-Kazar/89f523f15ec7744c8df204efbeb0fbf1262b2ca7. https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlSelect.html

Samarati, P., & De Vimercati, S. C. (2001). Access control: Policies, models, and mechanisms. Lecture Notes in Computer Science, 137–196. https://doi.org/10.1007/3-540-45608-2_3

Silberschatz, A., Korth, H. F., & Sudarshan, S. (2011). Database system concepts. McGraw-Hill Education.

Sug, H. (2020). A method for normalization of relation schema based on data to abide by the third normal form. https://www.semanticscholar.org/paper/A-Method-for-Normalization-of-Relation-Schema-Based-Sug/e4ac40b7355761c0694334f30fb97d7cdd1b2a0a. Update | CQL for Cassandra 3.0." https://docs.datastax.com/en/cql-oss/3.3/cql/cql_reference/cqlUpdate.html, 19, 216–225. https://doi.org/10.37394/23206.2020.19.20

Update documents – MongoDB manual v7.0. https://www.mongodb.com/docs/manual/tutorial/update-documents/

Walke, D., Micheel, D., Schallert, K., Muth, T., Broneske, D., Saake, G., & Heyer, R. (2023). The importance of graph databases and graph learning for clinical applications [Database]. Database, 2023, Article baad045. https://doi.org/10.1093/database/baad045

Wang, X., Wu, W., Wu, J., Chen, Y., Zrymiak, N., Qu, C., Flokas, L., Chow, G., Wang, J., Wang, T., Wu, E., & Zhou, Q. (2022). ConnectorX: Accelerating data loading from databases to dataframes. https://www.semanticscholar.org/paper/ConnectorX%3A-Accelerating-Data-Loading-From-to-Wang-Wu/2118fc6dcb70cac36c783800a4e5487899102476. Proceedings of the VLDB Endowment, 15(11), 2994–3003. https://doi.org/10.14778/3551793.3551847

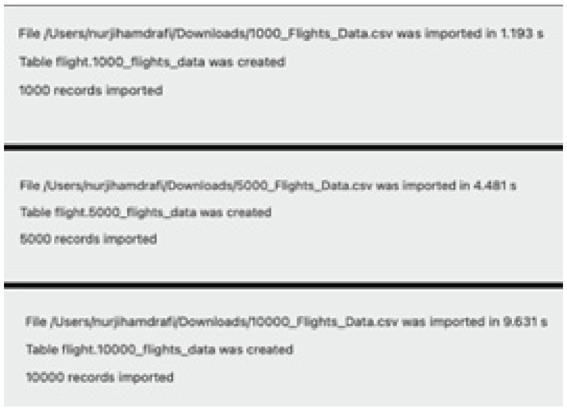
What is data integrity and why does it matter? (2021). Business insights blog, February 04. https://online.hbs.edu/blog/post/what-is-data-integrity

Zaniewicz, N., & Salamończyk, A. (2022). Comparison of MongoDB, Neo4j and ArangoDB databases using the developed data generator for NoSQL databases. https://www.semanticscholar.org/paper/Comparison-of-MongoDB%2C-Neo4j-and-ArangoDB-databases-Zaniewicz-Salamo%C5%84czyk/64f5b93e0320c46544b12f838df3413c5a682d38. Studia Informatica. System and Information Technology, 26(1), 61–72. https://doi.org/10.34739/si.2022.26.04

Cite this article: Ying CP, Min JCS, Jean GY, Xuan LZ, Rafi NNB, Sathishkumar VE, *et al*. Comparative Analysis of Relational and Non-Relational Database Models: A Case Study on Travel Booking Systems. Info Res Com. 2025;2(1):98-119.

Data creation

MySQL

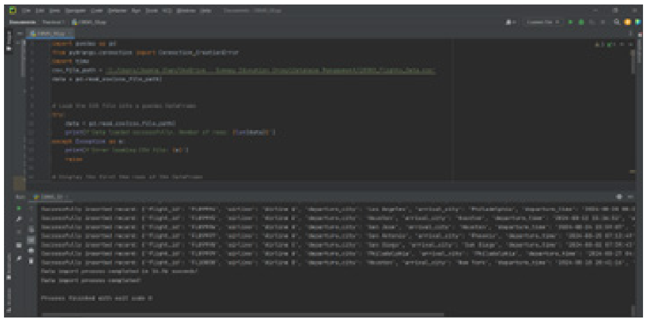


```
File /Users/nurjihamdrafi/Downloads/1000_Flights_Data.csv was imported in 1.193 s
Table flight.1000_flights_data was created
1000 records imported
```

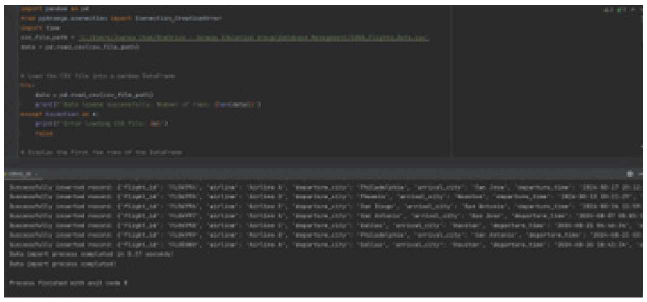
```
File /Users/nurjihamdrafi/Downloads/5000_Flights_Data.csv was imported in 4.481 s
Table flight.5000_flights_data was created
5000 records imported
```

```
File /Users/nurjihamdrafi/Downloads/10000_Flights_Data.csv was imported in 9.631 s
Table flight.10000_flights_data was created
10000 records imported
```

Arango DB

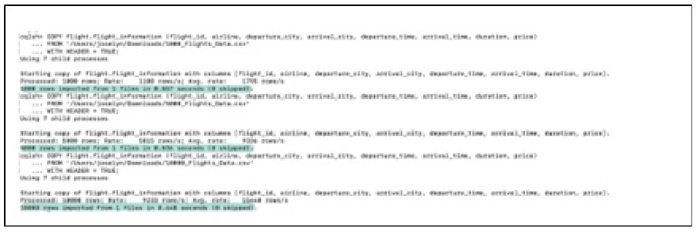


```
import pandas as pd
from pyArango.connection import Connection,CreationError
import time
csv_file_path = [illegible]
data = pd.read_csv(csv_file_path)

# Load the CSV file into a pandas DataFrame
try:
    data = pd.read_csv(csv_file_path)
    print(f"Data loaded successfully. Number of rows: {len(data)}")
except Exception as e:
    print(f"Error loading CSV file: {e}")
    raise

# Display the first few rows of the DataFrame

Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'Los Angeles', 'arrival_city': 'Philadelphia', 'departure_time': [illegible]
[illegible]
Data import process completed in [illegible] seconds.
Data import process completed!

Process finished with exit code 0
```



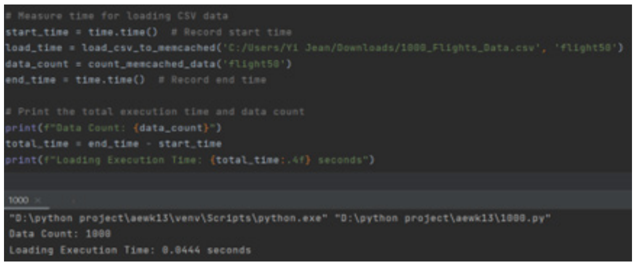
```
import pandas as pd
from pyArango.connection import Connection,CreationError
import time
csv_file_path = [illegible]
data = pd.read_csv(csv_file_path)

# Load the CSV file into a pandas DataFrame
try:
    data = pd.read_csv(csv_file_path)
    print(f"Data loaded successfully. Number of rows: {len(data)}")
except Exception as e:
    print(f"Error loading CSV file: {e}")
    raise

# Display the first few rows of the DataFrame

Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'Philadelphia', 'arrival_city': 'San Jose', 'departure_time': [illegible]
Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'Phoenix', 'arrival_city': 'Houston', 'departure_time': [illegible]
Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'San Diego', 'arrival_city': 'San Antonio', 'departure_time': [illegible]
Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'San Antonio', 'arrival_city': 'San Jose', 'departure_time': [illegible]
Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'Dallas', 'arrival_city': 'Houston', 'departure_time': [illegible]
Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'Philadelphia', 'arrival_city': 'San Antonio', 'departure_time': [illegible]
Successfully inserted record: {'flight_id': [illegible], 'airline': [illegible], 'departure_city': 'Dallas', 'arrival_city': 'Houston', 'departure_time': [illegible]
Data import process completed in [illegible] seconds.
Data import process completed!

Process finished with exit code 0
```

Apache Cassandra

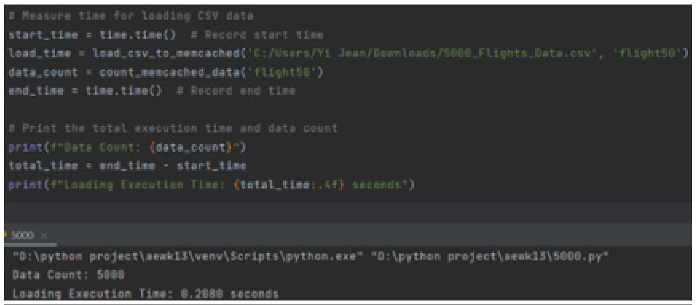


```
cqlsh> COPY flight.flight_information (flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, duration, price)
   ... FROM '/Users/joselyn/Downloads/1000_Flights_Data.csv'
   ... WITH HEADER = TRUE;
Using 7 child processes

Starting copy of flight.flight_information with columns [flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, duration, price].
Processed: 1000 rows; Rate: [illegible] rows/s; Avg. rate: [illegible] rows/s
1000 rows imported from 1 files in [illegible] seconds (0 skipped).
cqlsh> COPY flight.flight_information (flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, duration, price)
   ... FROM '/Users/joselyn/Downloads/5000_Flights_Data.csv'
   ... WITH HEADER = TRUE;
Using 7 child processes

Starting copy of flight.flight_information with columns [flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, duration, price].
Processed: 5000 rows; Rate: [illegible] rows/s; Avg. rate: [illegible] rows/s
5000 rows imported from 1 files in [illegible] seconds (0 skipped).
cqlsh> COPY flight.flight_information (flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, duration, price)
   ... FROM '/Users/joselyn/Downloads/10000_Flights_Data.csv'
   ... WITH HEADER = TRUE;
Using 7 child processes

Starting copy of flight.flight_information with columns [flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, duration, price].
Processed: 10000 rows; Rate: [illegible] rows/s; Avg. rate: [illegible] rows/s
10000 rows imported from 1 files in [illegible] seconds (0 skipped).
```

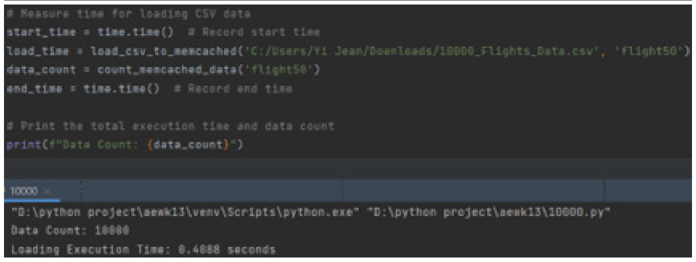
Memcached



```
# Measure time for loading CSV data
start_time = time.time()  # Record start time
load_time = load_csv_to_memcached('C:/Users/Yi Jean/Downloads/1000_Flights_Data.csv', 'flight50')
data_count = count_memcached_data('flight50')
end_time = time.time()  # Record end time

# Print the total execution time and data count
print(f"Data Count: {data_count}")
total_time = end_time - start_time
print(f"Loading Execution Time: {total_time:.4f} seconds")

1000

"D:\python project\aewk13\venv\Scripts\python.exe" "D:\python project\aewk13\1000.py"
Data Count: 1000
Loading Execution Time: 0.0444 seconds
```

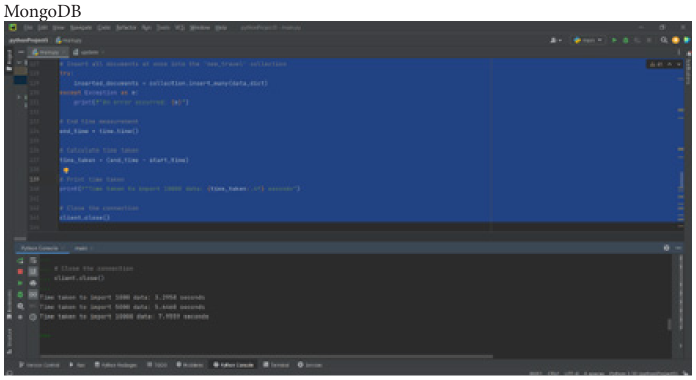


```
# Measure time for loading CSV data
start_time = time.time()  # Record start time
load_time = load_csv_to_memcached('C:/Users/Yi Jean/Downloads/5000_Flights_Data.csv', 'flight50')
data_count = count_memcached_data('flight50')
end_time = time.time()  # Record end time

# Print the total execution time and data count
print(f"Data Count: {data_count}")
total_time = end_time - start_time
print(f"Loading Execution Time: {total_time:.4f} seconds")

5000

"D:\python project\aewk13\venv\Scripts\python.exe" "D:\python project\aewk13\5000.py"
Data Count: 5000
Loading Execution Time: 0.2880 seconds
```



```
# Measure time for loading CSV data
start_time = time.time()  # Record start time
load_time = load_csv_to_memcached('C:/Users/Yi Jean/Downloads/10000_Flights_Data.csv', 'flight50')
data_count = count_memcached_data('flight50')
end_time = time.time()  # Record end time

# Print the total execution time and data count
print(f"Data Count: {data_count}")

10000

"D:\python project\aewk13\venv\Scripts\python.exe" "D:\python project\aewk13\10000.py"
Data Count: 10000
Loading Execution Time: 0.4888 seconds
```

MongoDB

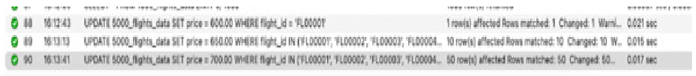


```
# Insert all documents at once into the 'new_travel' collection
try:
    inserted_documents = collection.insert_many(data_dict)
except Exception as e:
    print(f"An error occurred: {e}")

# End time measurement
end_time = time.time()

# Calculate time taken
time_taken = (end_time - start_time)

# Print time taken
print(f"Time taken to import 10000 data: {time_taken:.4f} seconds")

# Close the connection
client.close()

Python Console

    # Close the connection
    client.close()

Time taken to import 1000 data: 1.2958 seconds
Time taken to import 5000 data: 5.4448 seconds
Time taken to import 10000 data: 7.9559 seconds
```

Data Manipulation

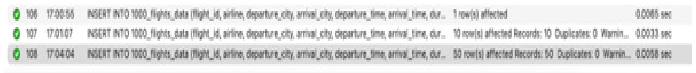
MySQL

Update



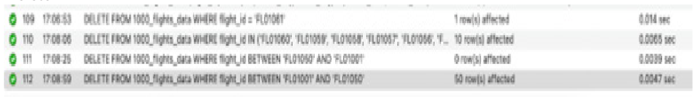
```
88  16:12:43  UPDATE 5000_flights_data SET price = 600.00 WHERE flight_id = 'FL00001'                                        1 row(s) affected Rows matched: 1  Changed: 1  Warni...  0.021 sec
89  16:13:13  UPDATE 5000_flights_data SET price = 650.00 WHERE flight_id IN ('FL00001', 'FL00002', 'FL00003', 'FL00004...  10 row(s) affected Rows matched: 10  Changed: 10  W...  0.015 sec
90  16:13:41  UPDATE 5000_flights_data SET price = 700.00 WHERE flight_id IN ('FL00001', 'FL00002', 'FL00003', 'FL00004...  50 row(s) affected Rows matched: 50  Changed: 50...     0.017 sec
```

Insert



```
106  17:00:55  INSERT INTO 1000_flights_data (flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, dur...  1 row(s) affected                                    0.0065 sec
107  17:01:07  INSERT INTO 1000_flights_data (flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, dur...  10 row(s) affected Records: 10  Duplicates: 0  Warnin...  0.0033 sec
108  17:04:04  INSERT INTO 1000_flights_data (flight_id, airline, departure_city, arrival_city, departure_time, arrival_time, dur...  50 row(s) affected Records: 50  Duplicates: 0  Warnin...  0.0058 sec
```

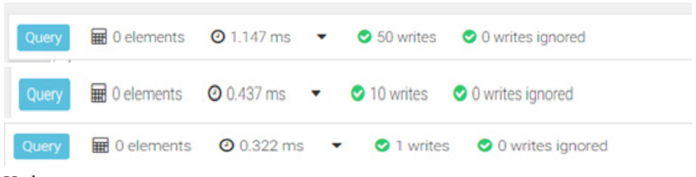
Delete



```
109  17:06:53  DELETE FROM 1000_flights_data WHERE flight_id = 'FL01061'                                                  1 row(s) affected   0.014 sec
110  17:08:06  DELETE FROM 1000_flights_data WHERE flight_id IN ('FL01060', 'FL01059', 'FL01058', 'FL01057', 'FL01056', 'F...  10 row(s) affected  0.0065 sec
111  17:08:25  DELETE FROM 1000_flights_data WHERE flight_id BETWEEN 'FL01050' AND 'FL01001'                                  0 row(s) affected   0.0039 sec
112  17:08:59  DELETE FROM 1000_flights_data WHERE flight_id BETWEEN 'FL01001' AND 'FL01050'                                  50 row(s) affected  0.0047 sec
```

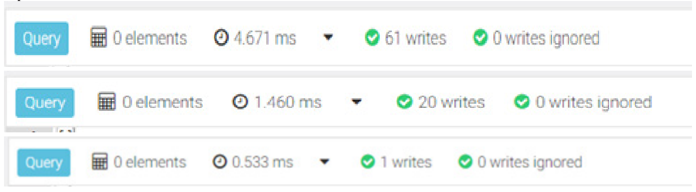
ArangoDB

Insert



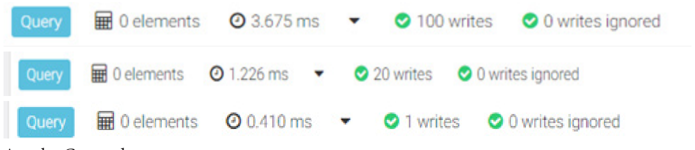
```
Query   0 elements   1.147 ms   50 writes   0 writes ignored
Query   0 elements   0.437 ms   10 writes   0 writes ignored
Query   0 elements   0.322 ms   1 writes   0 writes ignored
```

Update



```
Query   0 elements   4.671 ms   61 writes   0 writes ignored
Query   0 elements   1.460 ms   20 writes   0 writes ignored
Query   0 elements   0.533 ms   1 writes   0 writes ignored
```

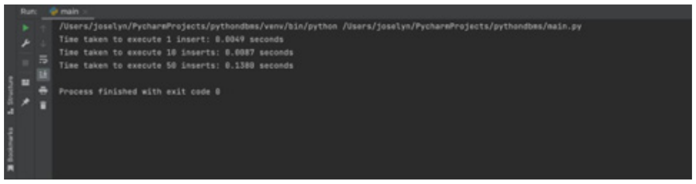
Delete



```
Query   0 elements   3.675 ms   100 writes   0 writes ignored
Query   0 elements   1.226 ms   20 writes    0 writes ignored
Query   0 elements   0.410 ms   1 writes     0 writes ignored
```

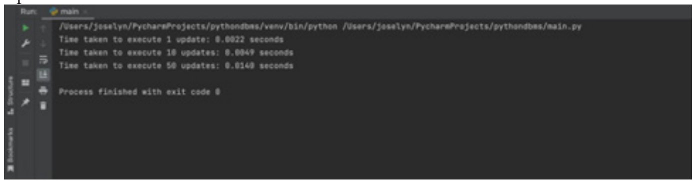
Apache Cassandra

Insert



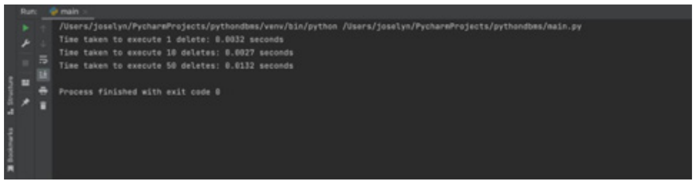
```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Time taken to execute 1 insert: 0.0049 seconds
Time taken to execute 10 inserts: 0.0087 seconds
Time taken to execute 50 inserts: 0.1380 seconds

Process finished with exit code 0
```

Update



```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Time taken to execute 1 update: 0.0022 seconds
Time taken to execute 10 updates: 0.0049 seconds
Time taken to execute 50 updates: 0.0140 seconds

Process finished with exit code 0
```

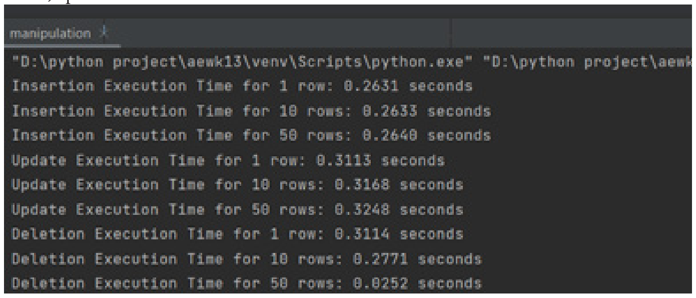
Delete



```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Time taken to execute 1 delete: 0.0032 seconds
Time taken to execute 10 deletes: 0.0027 seconds
Time taken to execute 50 deletes: 0.0132 seconds

Process finished with exit code 0
```

Memcached

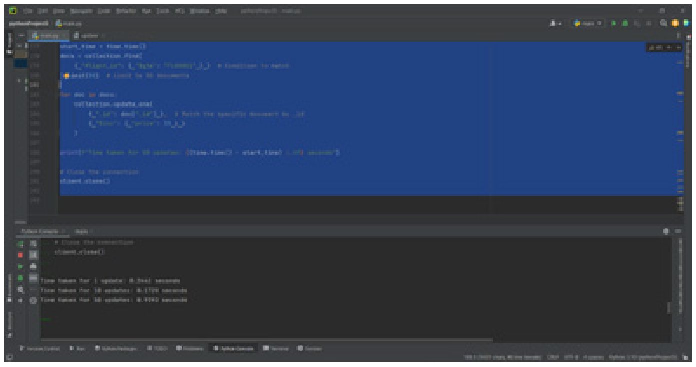
Insert, update delete



```
manipulation
"D:\python project\aewk13\venv\Scripts\python.exe" "D:\python project\aewk
Insertion Execution Time for 1 row: 0.2631 seconds
Insertion Execution Time for 10 rows: 0.2633 seconds
Insertion Execution Time for 50 rows: 0.2640 seconds
Update Execution Time for 1 row: 0.3113 seconds
Update Execution Time for 10 rows: 0.3168 seconds
Update Execution Time for 50 rows: 0.3248 seconds
Deletion Execution Time for 1 row: 0.3114 seconds
Deletion Execution Time for 10 rows: 0.2771 seconds
Deletion Execution Time for 50 rows: 0.0252 seconds
```

MongoDB

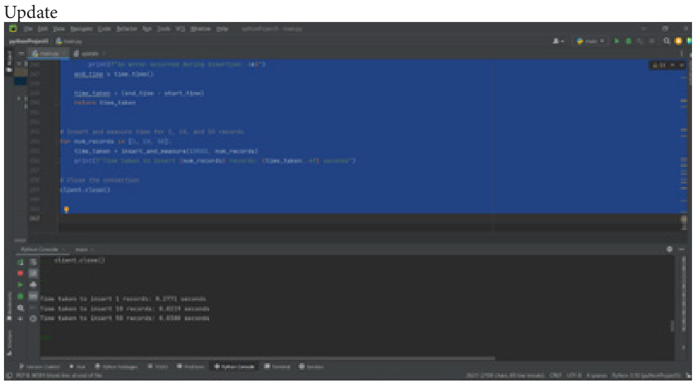
Insert



```
start_time = time.time()
docs = collection.find(
    {"flight_id": {"$gte": "FL00001"}}  # Condition to match
).limit(50)  # Limit to 50 documents

for doc in docs:
    collection.update_one(
        {"_id": doc["_id"]},  # Match the specific document by _id
        {"$inc": {"price": 10}}
    )

print(f"Time taken for 50 updates: {(time.time() - start_time):.4f} seconds")

# Close the connection
client.close()
```

```
Time taken for 1 update: 0.2442 seconds
Time taken for 10 updates: 0.1728 seconds
Time taken for 50 updates: 0.9393 seconds
```

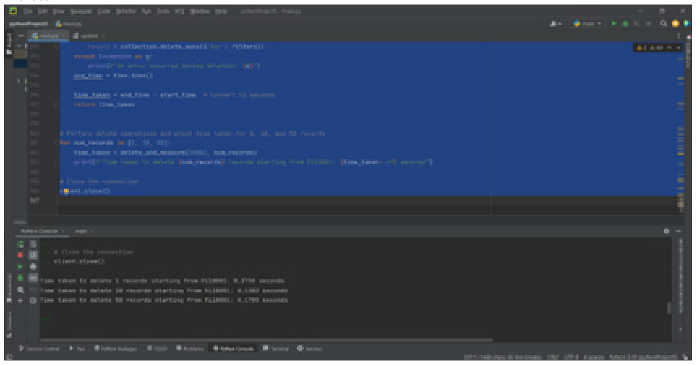
Update



```
        print(f"An error occurred during insertion: {e}")
    end_time = time.time()

    time_taken = (end_time - start_time)
    return time_taken


# Insert and measure time for 1, 10, and 50 records
for num_records in [1, 10, 50]:
    time_taken = insert_and_measure(10001, num_records)
    print(f"Time taken to insert {num_records} records: {time_taken:.4f} seconds")

# Close the connection
client.close()
```

```
Time taken to insert 1 records: 0.2771 seconds
Time taken to insert 10 records: 0.0217 seconds
Time taken to insert 50 records: 0.0398 seconds
```

Delete

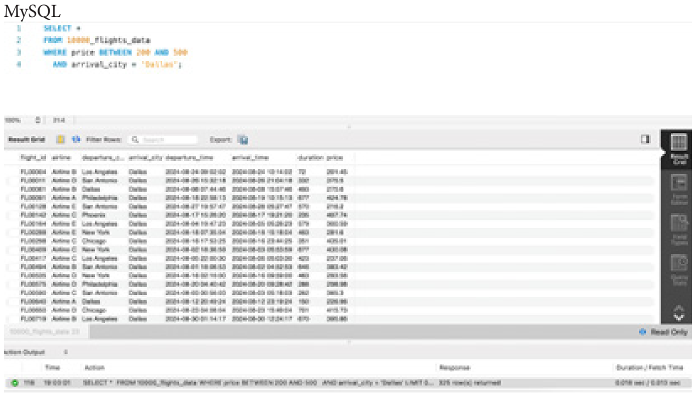


```
        result = collection.delete_many({"_id": {"$in": ids}})
    except Exception as e:
        print(f"An error occurred during deletion: {e}")
    end_time = time.time()

    time_taken = end_time - start_time  # Convert to seconds
    return time_taken


# Perform delete operations and print time taken for 1, 10, and 50 records
for num_records in [1, 10, 50]:
    time_taken = delete_and_measure(10001, num_records)
    print(f"Time taken to delete {num_records} records starting from FL10001: {time_taken:.4f} seconds")

# Close the connection
client.close()
```

```
Time taken to delete 1 records starting from FL10001: 0.3738 seconds
Time taken to delete 10 records starting from FL10001: 0.1302 seconds
Time taken to delete 50 records starting from FL10001: 0.1705 seconds
```

Data Retrieval

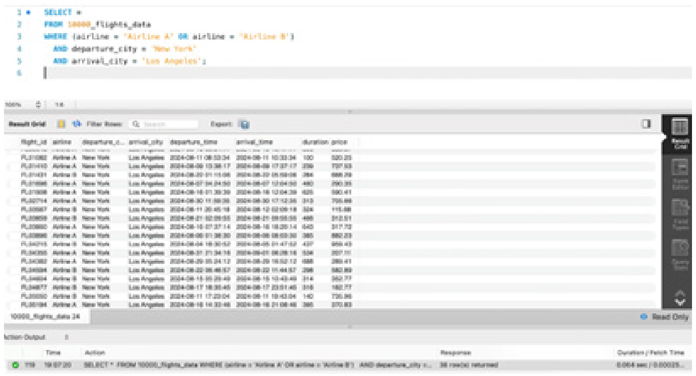
MySQL



```sql
SELECT *
FROM 10000_flights_data
WHERE price BETWEEN 200 AND 500
  AND arrival_city = 'Dallas';
```

flight_id	airline	departure_c...	arrival_city	departure_time	arrival_time	duration	price
FL00004	Airline B	Los Angeles	Dallas	2024-08-24 09:02:02	2024-08-24 10:14:02	72	281.45
FL00011	Airline D	San Antonio	Dallas	2024-08-26 15:32:18	2024-08-26 21:04:18	332	375.5
FL00061	Airline B	Dallas	Dallas	2024-08-08 07:44:46	2024-08-08 15:07:46	443	273.6
FL00091	Airline A	Philadelphia	Dallas	2024-08-18 22:58:13	2024-08-19 10:15:13	677	424.76
FL00128	Airline E	San Antonio	Dallas	2024-08-27 19:57:47	2024-08-28 05:27:47	570	218.2
FL00142	Airline C	Phoenix	Dallas	2024-08-17 15:26:20	2024-08-17 19:21:20	235	487.74
FL00164	Airline E	Los Angeles	Dallas	2024-08-04 19:47:23	2024-08-05 05:26:23	579	300.59
FL00268	Airline E	New York	Dallas	2024-08-18 07:35:04	2024-08-18 15:18:04	463	281.6
FL00298	Airline C	Chicago	Dallas	2024-08-16 17:52:25	2024-08-16 23:44:25	351	435.01
FL00409	Airline C	New York	Dallas	2024-08-02 18:36:59	2024-08-03 05:53:59	677	430.08
FL00417	Airline C	Los Angeles	Dallas	2024-08-05 22:00:30	2024-08-05 05:03:30	423	237.06
FL00484	Airline B	San Antonio	Dallas	2024-08-01 18:06:53	2024-08-02 04:52:53	646	383.42
FL00505	Airline D	New York	Dallas	2024-08-14 02:16:00	2024-08-14 09:59:00	463	293.56
FL00575	Airline D	Philadelphia	Dallas	2024-08-20 04:40:42	2024-08-20 09:28:42	288	298.98
FL00580	Airline C	San Antonio	Dallas	2024-08-03 00:56:03	2024-08-03 05:18:03	262	365.3
FL00640	Airline A	Dallas	Dallas	2024-08-12 20:49:24	2024-08-12 23:19:24	150	225.86
FL00650	Airline D	Chicago	Dallas	2024-08-23 04:08:04	2024-08-23 15:49:04	701	415.73
FL00719	Airline B	Los Angeles	Dallas	2024-08-30 01:14:17	2024-08-30 12:24:17	670	395.86

```
Action Output
    Time      Action                                                                                          Response              Duration / Fetch Time
118 19:03:01  SELECT * FROM 10000_flights_data WHERE price BETWEEN 200 AND 500 AND arrival_city = 'Dallas' LIMIT 0...  325 row(s) returned   0.018 sec / 0.013 sec
```

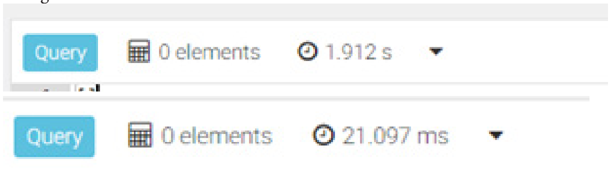


```sql
SELECT *
FROM 10000_flights_data
WHERE (airline = 'Airline A' OR airline = 'Airline B')
  AND departure_city = 'New York'
  AND arrival_city = 'Los Angeles';
```

flight_id	airline	departure_c...	arrival_city	departure_time	arrival_time	duration	price
FL01082	Airline A	New York	Los Angeles	2024-08-11 08:53:34	2024-08-11 10:33:34	100	520.25
FL01410	Airline A	New York	Los Angeles	2024-08-09 13:38:17	2024-08-09 17:37:17	239	737.53
FL01431	Airline B	New York	Los Angeles	2024-08-20 01:11:06	2024-08-20 05:59:06	284	688.29
FL01686	Airline A	New York	Los Angeles	2024-08-07 04:24:50	2024-08-07 12:04:50	460	290.35
FL01938	Airline A	New York	Los Angeles	2024-08-18 01:39:39	2024-08-18 12:04:39	625	590.41
FL02714	Airline A	New York	Los Angeles	2024-08-30 11:59:35	2024-08-30 17:12:35	313	705.66
FL03587	Airline B	New York	Los Angeles	2024-08-11 20:45:18	2024-08-12 02:09:18	324	115.68
FL03659	Airline B	New York	Los Angeles	2024-08-21 02:09:55	2024-08-21 09:55:55	466	312.51
FL03860	Airline A	New York	Los Angeles	2024-08-18 07:37:14	2024-08-18 18:20:14	643	317.72
FL03895	Airline A	New York	Los Angeles	2024-08-08 01:38:30	2024-08-08 08:03:30	385	882.23
FL04215	Airline B	New York	Los Angeles	2024-08-04 18:30:52	2024-08-05 01:47:52	437	959.43
FL04355	Airline A	New York	Los Angeles	2024-08-31 21:34:16	2024-09-01 08:28:16	534	207.11
FL04362	Airline A	New York	Los Angeles	2024-08-29 05:24:12	2024-08-29 16:52:12	688	280.41
FL04594	Airline B	New York	Los Angeles	2024-08-22 06:46:57	2024-08-22 11:44:57	298	582.89
FL04604	Airline B	New York	Los Angeles	2024-08-15 05:29:49	2024-08-15 10:43:49	314	352.77
FL04677	Airline B	New York	Los Angeles	2024-08-17 18:35:45	2024-08-17 23:01:45	316	182.77
FL05050	Airline B	New York	Los Angeles	2024-08-11 17:22:04	2024-08-11 19:42:04	140	720.86
FL05194	Airline A	New York	Los Angeles	2024-08-18 14:33:46	2024-08-18 21:08:46	395	370.83

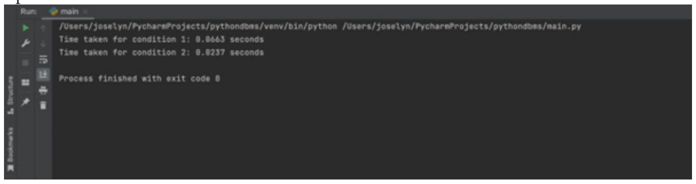
```
Action Output
    Time      Action                                                                                                  Response             Duration / Fetch Time
119 19:07:20  SELECT * FROM 10000_flights_data WHERE (airline = 'Airline A' OR airline = 'Airline B') AND departure_city =...  36 row(s) returned   0.004 sec / 0.00025...
```

ArangoDB



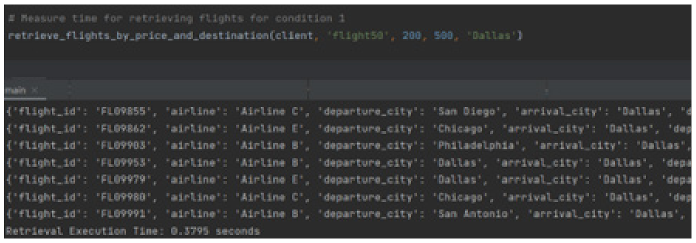
```
Query   0 elements   1.912 s
Query   0 elements   21.097 ms
```

Apache Cassandra



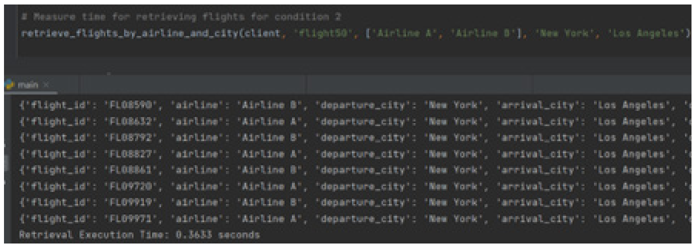
```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Time taken for condition 1: 0.0663 seconds
Time taken for condition 2: 0.0237 seconds

Process finished with exit code 0
```

Memcached



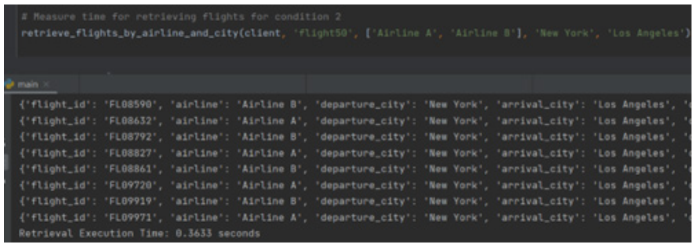
```
# Measure time for retrieving flights for condition 1
retrieve_flights_by_price_and_destination(client, 'flight50', 200, 500, 'Dallas')
```

```
{'flight_id': 'FL09855', 'airline': 'Airline C', 'departure_city': 'San Diego', 'arrival_city': 'Dallas', 'd
{'flight_id': 'FL09862', 'airline': 'Airline E', 'departure_city': 'Chicago', 'arrival_city': 'Dallas', 'dep
{'flight_id': 'FL09903', 'airline': 'Airline B', 'departure_city': 'Philadelphia', 'arrival_city': 'Dallas',
{'flight_id': 'FL09953', 'airline': 'Airline B', 'departure_city': 'Dallas', 'arrival_city': 'Dallas', 'depa
{'flight_id': 'FL09979', 'airline': 'Airline E', 'departure_city': 'Dallas', 'arrival_city': 'Dallas', 'depa
{'flight_id': 'FL09980', 'airline': 'Airline C', 'departure_city': 'Chicago', 'arrival_city': 'Dallas', 'dep
{'flight_id': 'FL09991', 'airline': 'Airline B', 'departure_city': 'San Antonio', 'arrival_city': 'Dallas',
Retrieval Execution Time: 0.3795 seconds
```



```
# Measure time for retrieving flights for condition 2
retrieve_flights_by_airline_and_city(client, 'flight50', ['Airline A', 'Airline B'], 'New York', 'Los Angeles')
```

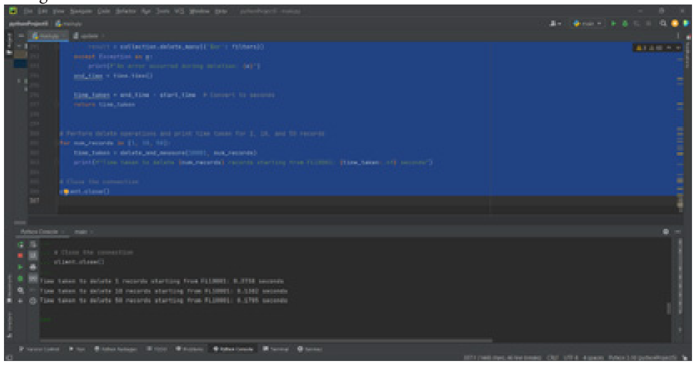
```
{'flight_id': 'FL08590', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08632', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08792', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08827', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08861', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL09720', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL09919', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL09971', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
Retrieval Execution Time: 0.3633 seconds
```



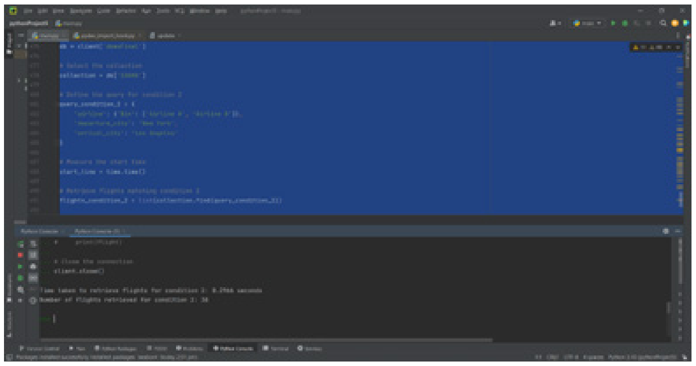
```
# Measure time for retrieving flights for condition 2
retrieve_flights_by_airline_and_city(client, 'flight50', ['Airline A', 'Airline B'], 'New York', 'Los Angeles')
```

```
{'flight_id': 'FL08590', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08632', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08792', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08827', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL08861', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL09720', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL09919', 'airline': 'Airline B', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
{'flight_id': 'FL09971', 'airline': 'Airline A', 'departure_city': 'New York', 'arrival_city': 'Los Angeles', '
Retrieval Execution Time: 0.3633 seconds
```

MongoDB



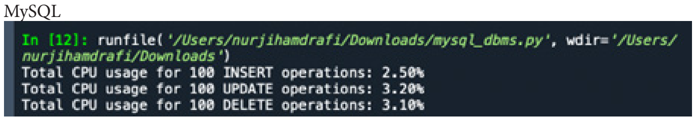
```
[illegible]
Time taken to delete 1 records starting from FL10001: [illegible] seconds
Time taken to delete 10 records starting from FL10001: [illegible] seconds
Time taken to delete 50 records starting from FL10001: [illegible] seconds
```



```
[illegible]
Time taken to retrieve flights for condition 2: 0.2966 seconds
Number of flights retrieved for condition 2: 38
```

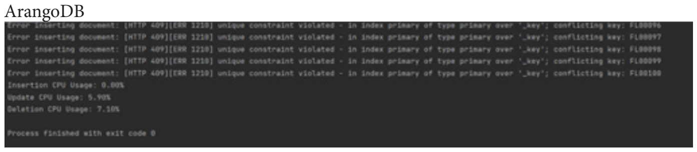
CPU usage

MySQL



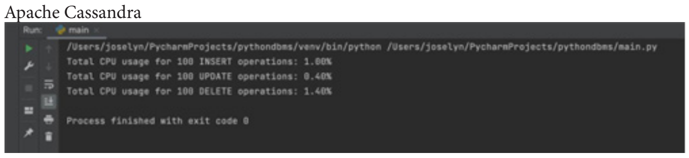
```
In [12]: runfile('/Users/nurjihamdrafi/Downloads/mysql_dbms.py', wdir='/Users/
nurjihamdrafi/Downloads')
Total CPU usage for 100 INSERT operations: 2.50%
Total CPU usage for 100 UPDATE operations: 3.20%
Total CPU usage for 100 DELETE operations: 3.10%
```

ArangoDB



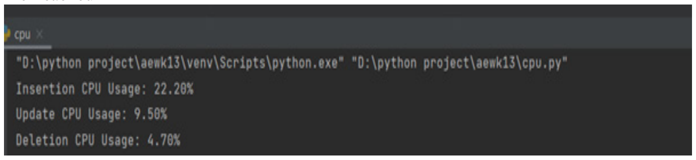
```
Error inserting document: [HTTP 409][ERR 1210] unique constraint violated - in index primary of type primary over '_key'; conflicting key: FL00096
Error inserting document: [HTTP 409][ERR 1210] unique constraint violated - in index primary of type primary over '_key'; conflicting key: FL00097
Error inserting document: [HTTP 409][ERR 1210] unique constraint violated - in index primary of type primary over '_key'; conflicting key: FL00098
Error inserting document: [HTTP 409][ERR 1210] unique constraint violated - in index primary of type primary over '_key'; conflicting key: FL00099
Error inserting document: [HTTP 409][ERR 1210] unique constraint violated - in index primary of type primary over '_key'; conflicting key: FL00100
Insertion CPU Usage: 0.00%
Update CPU Usage: 5.90%
Deletion CPU Usage: 7.10%

Process finished with exit code 0
```

Apache Cassandra



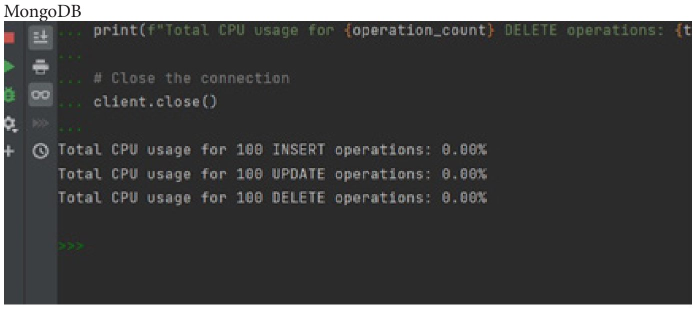
```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Total CPU usage for 100 INSERT operations: 1.00%
Total CPU usage for 100 UPDATE operations: 0.40%
Total CPU usage for 100 DELETE operations: 1.40%

Process finished with exit code 0
```

Memcached



```
"D:\python project\aewk13\venv\Scripts\python.exe" "D:\python project\aewk13\cpu.py"
Insertion CPU Usage: 22.20%
Update CPU Usage: 9.50%
Deletion CPU Usage: 4.70%
```

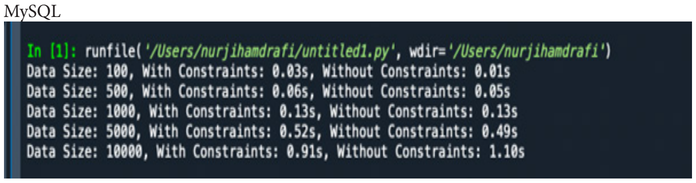
MongoDB



```
... print(f"Total CPU usage for {operation_count} DELETE operations: {t
...
... # Close the connection
... client.close()
...
Total CPU usage for 100 INSERT operations: 0.00%
Total CPU usage for 100 UPDATE operations: 0.00%
Total CPU usage for 100 DELETE operations: 0.00%

>>>
```

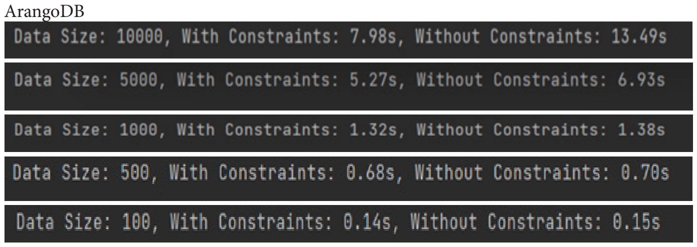
Data Integrity

MySQL



```
In [1]: runfile('/Users/nurjihamdrafi/untitled1.py', wdir='/Users/nurjihamdrafi')
Data Size: 100, With Constraints: 0.03s, Without Constraints: 0.01s
Data Size: 500, With Constraints: 0.06s, Without Constraints: 0.05s
Data Size: 1000, With Constraints: 0.13s, Without Constraints: 0.13s
Data Size: 5000, With Constraints: 0.52s, Without Constraints: 0.49s
Data Size: 10000, With Constraints: 0.91s, Without Constraints: 1.10s
```

ArangoDB



```
Data Size: 10000, With Constraints: 7.98s, Without Constraints: 13.49s
```

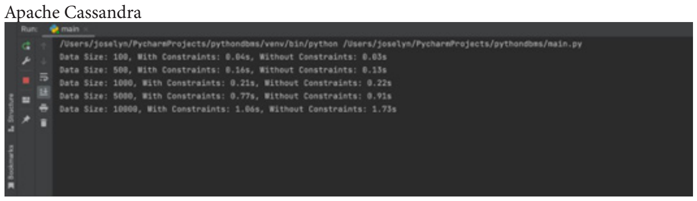
```
Data Size: 5000, With Constraints: 5.27s, Without Constraints: 6.93s
```

```
Data Size: 1000, With Constraints: 1.32s, Without Constraints: 1.38s
```

```
Data Size: 500, With Constraints: 0.68s, Without Constraints: 0.70s
```

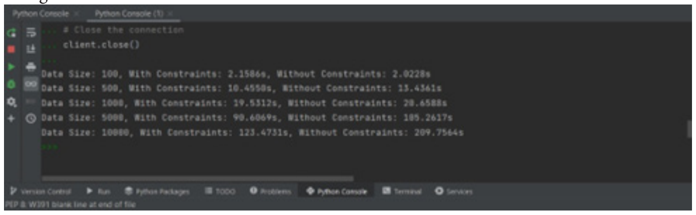
```
Data Size: 100, With Constraints: 0.14s, Without Constraints: 0.15s
```

Apache Cassandra



```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Data Size: 100, With Constraints: 0.04s, Without Constraints: 0.03s
Data Size: 500, With Constraints: 0.16s, Without Constraints: 0.13s
Data Size: 1000, With Constraints: 0.21s, Without Constraints: 0.22s
Data Size: 5000, With Constraints: 0.77s, Without Constraints: 0.91s
Data Size: 10000, With Constraints: 1.06s, Without Constraints: 1.73s
```

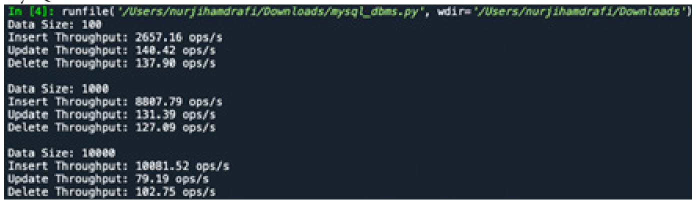
MongoDB



```
... # Close the connection
... client.close()
...
Data Size: 100, With Constraints: 2.1586s, Without Constraints: 2.0228s
Data Size: 500, With Constraints: 10.4558s, Without Constraints: 13.4361s
Data Size: 1000, With Constraints: 19.5312s, Without Constraints: 28.6588s
Data Size: 5000, With Constraints: 90.6069s, Without Constraints: 105.2617s
Data Size: 10000, With Constraints: 123.4731s, Without Constraints: 209.7564s
>>>
```

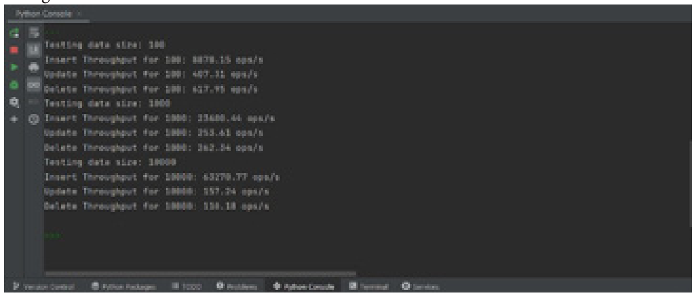
Scalability

MySQL



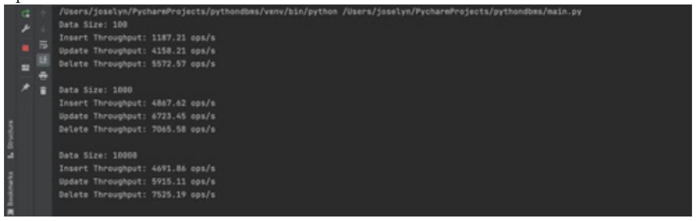
```
In [4]: runfile('/Users/nurjihamdrafi/Downloads/mysql_dbms.py', wdir='/Users/nurjihamdrafi/Downloads')
Data Size: 100
Insert Throughput: 2657.16 ops/s
Update Throughput: 140.42 ops/s
Delete Throughput: 137.90 ops/s

Data Size: 1000
Insert Throughput: 8807.79 ops/s
Update Throughput: 131.39 ops/s
Delete Throughput: 127.09 ops/s

Data Size: 10000
Insert Throughput: 10081.52 ops/s
Update Throughput: 79.19 ops/s
Delete Throughput: 102.75 ops/s
```

ArangoDB



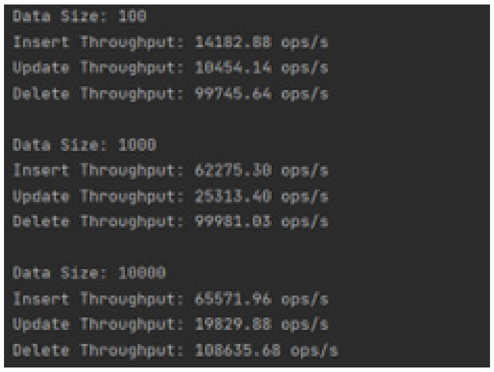
```
Testing data size: 100
Insert Throughput for 100: 8878.15 ops/s
Update Throughput for 100: 407.31 ops/s
Delete Throughput for 100: 617.95 ops/s
Testing data size: 1000
Insert Throughput for 1000: 23680.44 ops/s
Update Throughput for 1000: 253.61 ops/s
Delete Throughput for 1000: 362.34 ops/s
Testing data size: 10000
Insert Throughput for 10000: 63270.77 ops/s
Update Throughput for 10000: 157.24 ops/s
Delete Throughput for 10000: 158.18 ops/s

>>>
```

Apache Cassandra



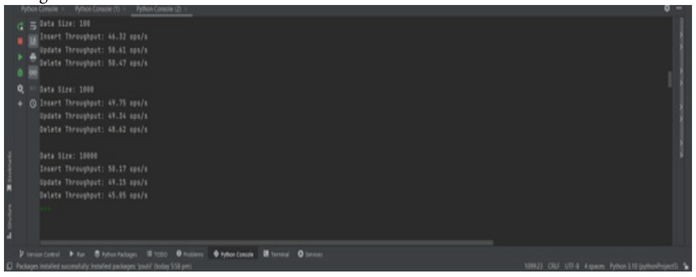
```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Data Size: 100
Insert Throughput: 1187.21 ops/s
Update Throughput: 4158.21 ops/s
Delete Throughput: 5572.57 ops/s

Data Size: 1000
Insert Throughput: 4867.62 ops/s
Update Throughput: 6723.45 ops/s
Delete Throughput: 7065.58 ops/s

Data Size: 10000
Insert Throughput: 4691.86 ops/s
Update Throughput: 5915.11 ops/s
Delete Throughput: 7525.19 ops/s
```

Memcached



```
Data Size: 100
Insert Throughput: 14182.88 ops/s
Update Throughput: 10454.14 ops/s
Delete Throughput: 99745.64 ops/s

Data Size: 1000
Insert Throughput: 62275.30 ops/s
Update Throughput: 25313.40 ops/s
Delete Throughput: 99981.03 ops/s

Data Size: 10000
Insert Throughput: 65571.96 ops/s
Update Throughput: 19829.88 ops/s
Delete Throughput: 108635.68 ops/s
```

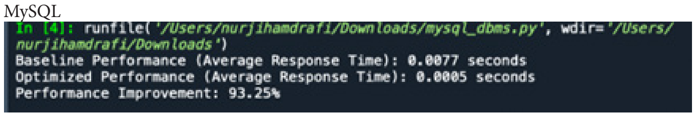
MongoDB



```
Data Size: 100
Insert Throughput: 46.32 ops/s
Update Throughput: 50.61 ops/s
Delete Throughput: 50.47 ops/s

Data Size: 1000
Insert Throughput: 49.75 ops/s
Update Throughput: 49.34 ops/s
Delete Throughput: 48.62 ops/s

Data Size: 10000
Insert Throughput: 50.17 ops/s
Update Throughput: 49.15 ops/s
Delete Throughput: 45.05 ops/s
>>>
```

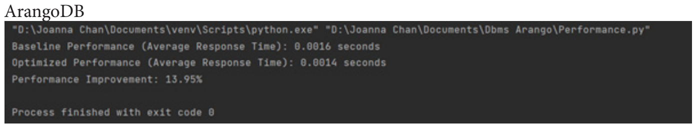
Performance Improvement

MySQL



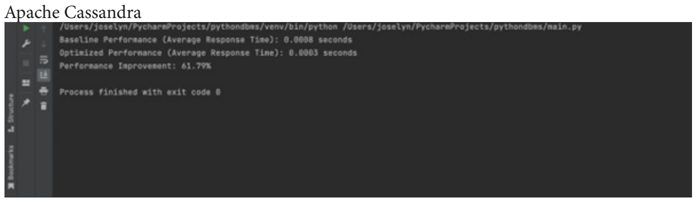
```
In [4]: runfile('/Users/nurjihamdrafi/Downloads/mysql_dbms.py', wdir='/Users/
nurjihamdrafi/Downloads')
Baseline Performance (Average Response Time): 0.0077 seconds
Optimized Performance (Average Response Time): 0.0005 seconds
Performance Improvement: 93.25%
```

ArangoDB



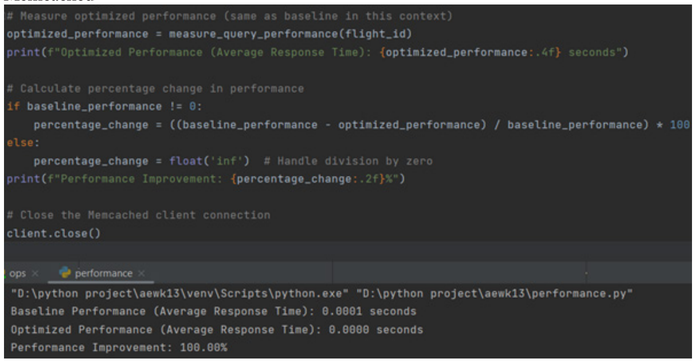
```
"D:\Joanna Chan\Documents\venv\Scripts\python.exe" "D:\Joanna Chan\Documents\Dbms Arango\Performance.py"
Baseline Performance (Average Response Time): 0.0016 seconds
Optimized Performance (Average Response Time): 0.0014 seconds
Performance Improvement: 13.95%

Process finished with exit code 0
```

Apache Cassandra



```
/Users/joselyn/PycharmProjects/pythondbms/venv/bin/python /Users/joselyn/PycharmProjects/pythondbms/main.py
Baseline Performance (Average Response Time): 0.0008 seconds
Optimized Performance (Average Response Time): 0.0003 seconds
Performance Improvement: 61.79%

Process finished with exit code 0
```

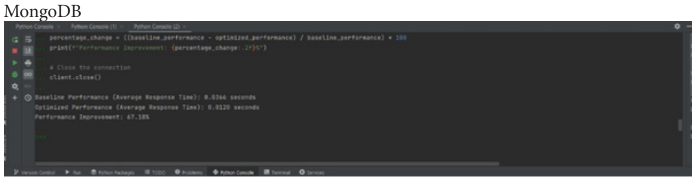
Memcached



```
# Measure optimized performance (same as baseline in this context)
optimized_performance = measure_query_performance(flight_id)
print(f"Optimized Performance (Average Response Time): {optimized_performance:.4f} seconds")

# Calculate percentage change in performance
if baseline_performance != 0:
    percentage_change = ((baseline_performance - optimized_performance) / baseline_performance) * 100
else:
    percentage_change = float('inf')  # Handle division by zero
print(f"Performance Improvement: {percentage_change:.2f}%")

# Close the Memcached client connection
client.close()
```

```
"D:\python project\aewk13\venv\Scripts\python.exe" "D:\python project\aewk13\performance.py"
Baseline Performance (Average Response Time): 0.0001 seconds
Optimized Performance (Average Response Time): 0.0000 seconds
Performance Improvement: 100.00%
```

MongoDB



```
... percentage_change = ((baseline_performance - optimized_performance) / baseline_performance) * 100
... print(f"Performance Improvement: {percentage_change:.2f}%")
...
... # Close the connection
... client.close()
...
Baseline Performance (Average Response Time): 0.0366 seconds
Optimized Performance (Average Response Time): 0.0120 seconds
Performance Improvement: 67.18%

>>>
```